Volume 15 Number 3

ACTA CYBERNETICA

Szeged, 2002

L. Budach
University of Postdam
Department of Computer Science
Am Neuen Palais 10
14415 Postdam, Germany

H. Bunke
Universität Bern
Institut für Informatik und
angewandte Mathematik
Längass strasse 51.
CH-3012 Bern, Switzerland

B. Courcelle
Universitè Bordeaux-1
LaBRI, 351 Cours de la Libèration
33405 TALENCE Cedex
France

J. Demetrovics
MTA SZTAKI
Budapest, Lágymányosi u. 11.
H-1111 Hungary

B. Dömölki
IQSOFT
Budapest, Teleki Blanka u. 15-17.
H-1142 Hungary

J. Engelfriet
Leiden University
LIACS
P.O. Box 9512, 2300 RA Leiden
The Netherlands

Z. Ésik
University of Szeged
Department of Foundations of
Computer Science
Szeged, Aradi vértanúk tere 1.
H-6720 Hungary

L. Lovász
Eötvös Loránd University
Department of Computer Science
Budapest, Kecskeméti u. 10-12.
H-1053 Hungary

G. Păun
Institute of Mathematics
Romanian Academy
P.O.Box 1-764, RO-70700
Bucuresti, Romania

A. Prékopa
Eötvös Loránd University
Department of Operations Research
Budapest, Kecskeméti u. 10-12.
H-1053 Hungary

A. Salomaa
University of Turku
Department of Mathematics
SF-20500 Turku 50, Finland

L. Varga
Eötvös Loránd University
Department of General Computer Science
Budapest, Pázmány Péter sétány 1/c.
H-1117 Hungary

H. Vogler
Dresden University of Technology
Department of Computer Science
Foundations of Programming
D-01062 Dresden, Germany

G. Wöginger
Department of Matematics
University of Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands

Acta Cybernetica **15** (2002) 321–325.

A form of the Zermelo—von Neumann theorem under minimal assumptions*

B. Csákány †

To the memory of L. Kalmár (1905-1976)

Abstract

A simple and general version of the classical result in the title is formulated and proved in the form of a proposition concerning formal languages.

The fundamental game-theoretical theorem of von Neumann asserts that in a two-player zero-sum game both players have optimal (possibly mixed) strategies. A stronger statement is true for *chesslike games*, i.e. discrete finite games in which there are no chance moves, and there is complete information for both players. In such games, either one of the two players has a pure winning strategy or both players have pure safe strategies. We call this fact the Zermelo—von Neumann theorem, as Zermelo was the first to state an equivalent claim in [15], although he did not use the notion of a *strategy*, which was introduced and developed later in works of such pioneers as Borel [4], Steinhaus [14], von Neumann [10], and Kalmár [7]. In this note we prove a simple and general version of the Zermelo—von Neumann theorem. Here it takes the form of an assertion on formal languages, with no assumption on chance moves or complete information. The proof utilizes an idea of Fremlin [5] which dates back to the solution of the game Nim by Bouton [2]. The title alludes to the title of an article of Kalmár ([8]) in which a simple and general form of Gödel's incompleteness theorem is proposed and proved. Note that a fine analysis of interconnections between [15], [7], and a closely related article [9] by D. König may be found in a recent survey paper of Schwalbe and Walker ([13]), in which the definitive formulation of the Zermelo—von Neumann theorem is convincingly credited to Kalmár.

As usual, generators, elements and subsets of free monoids $F(P)$ will be called *letters*, *words*, and *languages*, respectively. A word w_1 is a *prefix* of the word w if $w = w_1 w_2$ for some word w_2; we write $w_1 \leq w$ in this case. The *prefix closure* of a language L is the set of all prefixes of all words in L. We say that L is *complete*, if

*The support of the Hungarian National Foundation for Scientific Research (OTKA) under Grant No. T026243 is gratefully acknowledged. Thanks are due to an anonymous referee for valuable suggestions and remarks.

†Bolyai Institute, University of Szeged, H-6720 Szeged, Aradi vértanúk tere 1, Hungary, E-mail: csakany@math.u-szeged.hu

every infinite chain $w_1 \leq w_2 \leq \ldots$ of elements of the prefix closure of L stabilizes (i.e., there exists an i such that $w_i = w_{i+1} = \ldots$). A prefix w_1 of w is *proper* if $w_1 \neq w$. A language L is *prefix-free* if no proper prefix of $w \in L$ is in L (cf., e.g., [6], [12], where such languages are called prefix codes). Here by a *game* we shall mean a nonempty complete prefix-free language $L \subseteq F(P)$ trisected into pairwise disjoint sets L_N, L_M, and L_T. For games $L \subseteq F(P)$ we adopt the following terminology: the letters (i.e., the elements of P) are the *positions*, all nonempty prefixes of the words in L are the *states*, and all pairs (s_1, s_2) of states such that there exist a position p with $s_2 = s_1 p$ are the *moves* of L. The words in L are the *terminal states*, the one-element prefixes of them are the *initial states* of the game L; finally, the words in L_N, L_M, and L_T are the *normal*, *misère*, and *tie* terminal states (cf. [1]), respectively. We also refer to terminal states of L as *L-games*.

The rationale of this terminology is that whenever two persons (say, White and Black) play a common finite discrete game G (as Nim, Chess, Go, card games, etc.), the whole process of playing—i.e., the G-game—is fully determined by the sequence $g = p_1 \ldots p_n$ of subsequent positions, and every move of G consists of choosing a further position p_{i+1} to continue a prefix $p_1 \ldots p_i$ $(1 \leq i < n)$ of g, according, of course, to the rules of the considered game. For several simple games (e.g., for Nim) the set of options depends only upon p_i. However, it may depend upon the parity of i, and, more generally, upon each position in $p_1 \ldots p_i$. This is the case, e.g., in Chess, in virtue of some special rules such as castling, en passant capturing, and the threefold repetition rule that prevents infinite games of Chess. Thus, we can consider every game $L \subseteq F(P)$ as an abstract form of a concrete two-player discrete game $\mathcal{L}$ with possible positions $p \in P$. The rule of moves of $\mathcal{L}$ is implicit in the set of all pairs of states of form $(p_1 \ldots p_i, p_1 \ldots p_i p_{i+1})$. As L is complete, this rule excludes the possibility of an infinite sequence of moves in $\mathcal{L}$; i.e., $\mathcal{L}$ is a finite game. The idea of considering states rather than positions goes back to the article [7], in which Kalmár introduced the script form of a game (Schriftspiel). The result of $\mathcal{L}$ is encoded into the components L_N, L_M, L_T of L: for $g \in L$, $g \in L_N$ means that the player unable to move *loses* (as in Nim), $g \in L_M$ indicates that he/she *wins* (e.g., L_M is empty if L stands for Chess), and $g \in L_T$ means that the L-game g ends in a *tie*.

As White and Black move alternately, White always moves *from* a state of odd length. Hence we call such states *White states*, and states of even length will be called *Black states* (including terminal states in both cases). Let S_W and S_B stands for the set of all White states, resp. Black states. Clearly, in an L-game g White wins iff $g \in L_N \cap S_B$ or $g \in L_M \cap S_W$, and Black wins iff $g \in L_N \cap S_W$ or $g \in L_M \cap S_B$. We define a *strategy of White* as a mapping w of the set of nonterminal White states into the set of Black states; similarly, a *strategy of Black* is a mapping $b : S_B \setminus L \to S_W$. Given a one-element word p_1 which is an initial state of L, any pair (w, b) of strategies determines a sequence

$$g(p_1, w, b) = p_1 \; w(p_1) \; b(w(p_1)) \; w(b(w(p_1))) \; b(w(b(w(p_1)))) \; \ldots$$

which, due to the completeness of L, cannot be infinite. Thus, $g(p_1, w, b)$ is an L-game with initial state p_1. A strategy w_0 of White is called a *winning strat-*

egy at p_1 if, for any strategy b of Black, White wins the game $g(p_1, w_0, b)$, and, correspondingly, a strategy b_0 of Black is winning at p_1 if, for any strategy w of White, Black wins the game $g(p_1, w, b_0)$. Finally, we call a strategy w_1 of White a *safe strategy at* p_1 if, for any strategy b of Black, either White wins $g(p_1, w_1, b)$ or $g(p_1, w_1, b)$ ends in a tie; a safe strategy of Black is defined similarly. We prove the following:

Given a game L *and an initial state* p_1 *of* L, *either one of Black and White has a winning strategy at* p_1 *or both of them have safe strategies at* p_1.

Consider a game L and let S be the set of all states of L. Call a triple (R_1, R_2, R_x), consisting of disjoint subsets of S, *regular* if it meets the following requirements:

(1) If $s \in R_1$ and (s, s') is a move, then $s' \in R_2$.
(2) If $s \in R_2$ and s is not a terminal state, then there exists a move (s, s') with $s' \in R_1$.
(3) If $s \in R_x$ and (s, s') is a move, then $s' \in R_2 \cup R_x$.
(4) If $s \in R_x$ and s is not a terminal state, then there exists a move (s, s') with $s' \in R_x$.

E.g., (L_N, L_M, L_T) is regular. The set of all regular triples is partially ordered by the rule

$$(R_1, R_2, R_x) \leq (R_1', R_2', R_x') \quad \text{iff} \quad R_1 \subseteq R_1',\ R_2 \subseteq R_2',\ R_x \subseteq R_x'.$$

Clearly, if $(R_1^\alpha, R_2^\alpha, R_x^\alpha)$ is a chain of regular triples, then $(\bigcup_\alpha R_1^\alpha, \bigcup_\alpha R_2^\alpha, \bigcup_\alpha R_x^\alpha)$ is regular, too. Hence there exists a maximal regular triple $(Q_1, Q_2, Q_x) \geq (N, M, T)$. We show that $Q = Q_1 \cup Q_2 \cup Q_x = S$. Suppose not. No states in $S \setminus Q$ are terminal. Therefore, as L is complete, there is a state $s \in S \setminus Q$ with the property that if (s, s') is a move, then $s' \in Q$. If all such s' are in Q_2, then $(Q_1 \cup \{s\}, Q_2, Q_x)$ is regular; if there is such an s' in Q_1, then $(Q_1, Q_2 \cup \{s\}, Q_x)$ is regular; and $(Q_1, Q_2, Q_x \cup \{s\})$ is regular in the remaining case, contradicting the maximality of (Q_1, Q_2, Q_x).

Define a function u on the set $S \setminus L$ of nonterminal states by choosing for $u(s)$

some $s' \in Q_2$ such that (s, s') is a move, if $s \in Q_1$,
some $s' \in Q_1$ such that (s, s') is a move, if $s \in Q_2$,
some $s' \in Q_x$ such that (s, s') is a move, if $s \in Q_x$.

Such a function exists by the axiom of choice and the definition of regular triples. Suppose that the initial state p_1 is in Q_1. Denote the restriction of u to $S_B \setminus L$ by u_1. Then u_1 is a strategy of Black. Consider an arbitrary strategy w of White. The L-game $g_1 = g(p_1, w, u_1)$ is either a White state in L_N or a Black state in L_M, i.e., Black wins in g_1. Thus, u_1 is a winning strategy at p_1 for Black. If $p_1 \in Q_2$, then similarly we obtain that the restriction of u to $S_W \setminus L$ is a winning strategy at p_1 for White. Finally, if $p_1 \in Q_x$, then these restrictions are safe strategies at p_1 for Black and White, respectively.

Note that the standard proof (see, e.g., [11] and [3]) goes by induction on the maximal length $n(p_1)$ of games with initial state p_1. The finiteness of a game

does not imply the existence of such an $n(p_1)$; this latter is a slight additional requirement on the game, whose fulfilment is usually guaranteed by postulating that the number of possible moves is finite in every state. This assumption is superfluous under our treatment. A simple example of a finite discrete game with no upper bound $n(p_1)$ on the number of possible consecutive moves from the initial state is the following. Two players place congruent coins onto a centrally symmetric table alternately; however, at most once during a game, instead of placing a coin, a player may choose to reduce arbitrarily but equally the size of all coins to be placed further on. The last player to move wins.

References

[1] E. R. Berlekamp, J. H. Conway, R. K. Guy, *Winning Ways, Vol. 1*, Academic Press, London, etc., 1982.

[2] A. L. Bouton, Nim, a game with a complete mathematical theory, *Annals of Math.*, **3**(1902), 35-39.

[3] D. Blackwell, M. A. Girshick, *Theory of Games and Statistical Decisions*, Wiley, New York, 1954.

[4] É. Borel, Sur les jeux où interviennent l'hasard et l'habileté des joueurs, *Théorie des Probabilités*, Paris, 1924; 202-224.

[5] D. Fremlin, Well-founded games, *Eureka*, **36**(1973), 33-37.

[6] G. A. Jones, J. M. Jones, *Information and Coding Theory*, Springer, London, 2000.

[7] L. Kalmár, Zur Theorie der abstrakten Spiele, *Acta Sci. Math. (Szeged)*, **4**(1928), 65-85.

[8] L. Kalmár, Une forme du théorème de Gödel sous des hypothèses minimales, *Comptes Rendus Acad. Sci.*, **229**(1949), 963-965.

[9] D. König, Über eine Schlussweise aus dem Endlichen ins Unendliche, *Acta Sci. Math. (Szeged)*, **3**(1927), 121-130.

[10] J. von Neumann, Zur Theorie der Gesellschaftsspiele, *Math. Ann.*, **100**(1928), 295-320.

[11] J. von Neumann, O. Morgenstern, *Theory of Games and Economic Behavior*, Princeton Univ. Press, Princeton, 1944.

[12] J. E. Pin, *Variétés de langages formels*, Masson, Paris, 1984.

[13] U. Schwalbe, P. Walker, Zermelo and the Early History of Game Theory, *Games and Economic Behavior*, **34**(2001), 123-137.

[14] H. Steinhaus, Definitions for a theory of games and pursuits, *Myśl Akademicka*, 1925 (in Polish); *Naval Res. Log. Quarterly*, **7**(1960), 105-108.

[15] E. Zermelo, Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels, *Proc. of the Fifth Congress of Mathematicians, Vol. 2*, Cambridge, 1913; 501-504.

Received August, 2001

Acta Cybernetica **15** (2002) 327–338.

Automaton Theory Approach for Solving Modified PNS Problems*

B. Imreh†

To Professor Masami Ito on his 60th birthday

Abstract

In this paper a modified version of the Process Network Synthesis (PNS) problem is studied. By using an automaton theoretical approach, a procedure for finding an optimal solution of this modified PNS problem is presented.

Introduction

The Process Network Synthesis (PNS for short) problem can be considered as a particular process design optimization. For this design, a set of the available operating units is given and each operating unit has a positive weight. Moreover, two distinguished sets of materials, the sets of raw materials and required products are also given. We are to find a minimum-weight process, consisting of the available operating units, which produces the required products from the raw materials. The corresponding processes from structural point of view can be identified by particular bipartite graphs satisfying some conditions. Such conditions are established in [4] and [5]. The bipartite graphs satisfying these conditions are called *solution-structures* and they can be considered as generalized feasible processes. This generalization means that we consider the processes in dynamic sense when we do not require the executability of processes. Therefore, a solution-structure may represent a non-executable process where by the *executability* of a process we mean that there exists such a scheduling of its operating units that the process can be performed in accordance with this scheduling. Here, by introducing a new condition for the bipartite graphs, we modify the original problem, concerning the generalized feasible processes, to such one whose feasible solutions represent exactly the executable feasible processes. For solving this modified problem, we extend the idea of [8]. Namely, for every instance of the modified problem, we define such an

*This work has been supported by the Hungarian National Foundation for Scientific Research, Grant T037258.

†Department of Informatics, University of Szeged, Árpád tér 2, H-6720 Szeged, Hungary

automaton that an optimal solution can be found by performing a shortest path method in the weighted transition graph of this automaton.

The paper is organized as follows. In Section 1, we recall the PNS problem and introduce its modified version, moreover, we recall some necessary notions and notation on automata. Then, in Section 2, the automaton theoretical approach and a procedure for finding an optimal solution are presented.

1 Preliminaries

Since the description of the original PNS problem can be found in more works (see *e.g.* [4], [5], [6], and [7]), we recall only the necessary definitions here. In the combinatorial approach, the structure of a process can be described by the process graph (*cf.* [5]) defined as follows.

Let M be a finite nonempty set, the set of the *materials*, and let $\emptyset \neq O \subseteq \wp'(M) \times \wp'(M)$ with $M \cap O = \emptyset$, where $\wp'(M)$ denotes the set of all nonempty subsets of M. The elements of O are called *operating units* and for any operating unit $u = (C, D) \in O$, C and D are called the *set of the input and output materials* of u, respectively. The pair (M, O) is called a *process graph*. The set of vertices of (M, O) is $M \cup O$, and the set of arcs is $E = E_1 \cup E_2$, where $E_1 = \{(x, u) : u = (C, D) \in O \ \& \ x \in C\}$ and $E_2 = \{(u, x) : u = (C, D) \in O \ \& \ x \in D\}$. If there are vertices $x_1, x_2, ..., x_n$, such that $(x_1, x_2), (x_2, x_3), \ldots, (x_{n-1}, x_n)$ are arcs of (M, O), then the path belonging to these arcs is denoted by $path[x_1, x_n]$. Let the process graphs $(\bar{M}, \bar{O})$ and (M, O) be given. $(\bar{M}, \bar{O})$ is called a *subgraph* of (M, O), if $\bar{M} \subseteq M$ and $\bar{O} \subseteq O$.

For any $\bar{O} \subseteq O$, let us define the following functions on $\bar{O}$:

$$mat^{in}(\bar{O}) = \bigcup_{(C,D)\in\bar{O}} C, \qquad mat^{out}(\bar{O}) = \bigcup_{(C,D)\in\bar{O}} D,$$

and

$$mat(\bar{O}) = mat^{in}(\bar{O}) \cup mat^{out}(\bar{O}).$$

Now, we can define the instances of the process design problem as follows. By an *instance* of the process design problem we mean a quartet $\mathbf{M} = (M, O, P, R)$, where M, O, P, R are finite sets, M is the set of the *available materials*, $\emptyset \neq P \subseteq M$ is the set of the *desired products*, $R \subseteq M$ is the set of the *raw materials*, and $\emptyset \neq O \subseteq \wp'(M) \times \wp'(M)$ is the set of the *available operating units*. It is supposed that $P \cap R = \emptyset$ and $M \cap O = \emptyset$. We are to design a process from structural point of view which produces the given set P of the required products from the given set R of the raw materials by using some available operating units.

Let us observe that the process graph (M, O) describes the interconnections among the operating units of O. Furthermore, every generalized feasible process

corresponds to a subgraph of (M, O). Consequently, we can determine the generalized feasible processes by examining the corresponding subgraphs of (M, O). If we do not consider further constraints such as material balance, then the subgraphs of (M, O) which can be assigned to the generalized feasible processes have common combinatorial properties. Such properties are established in [4] and [5]. A subgraph $(\bar{M}, \bar{O})$ of (M, O) is called a *solution-structure* of $(M, O, P, R,)$ if the following conditions are satisfied:

$(A1)$ $P \subseteq \bar{M}$,

$(A2)$ $\forall x \in \bar{M}$, $x \in R \Leftrightarrow$ no (u, x) arc in the process graph $(\bar{M}, \bar{O})$,

$(A3)$ $\forall u \in \bar{O}$, $\exists$ *path*$[u, x]$ with $x \in P$,

$(A4)$ $\forall x \in \bar{M}$, $\exists (C, D) \in \bar{O}$ such that $x \in C \cup D$.

The set of the solution-structures of $\mathbf{M}$ is denoted by $S(M, O, P, R,)$ or $S(\mathbf{M})$. We shall use the following observation which can be easily proved.

Remark 1. *If $(\bar{M}, \bar{O})$ is a solution-structure of* $\mathbf{M}$, *then $\bar{M} = mat(\bar{O})$, and hence, $\bar{O}$ determines the solution-structure $(\bar{M}, \bar{O})$ uniquely.*

Let us now consider an instance of the process design problem in which every operating unit has a positive real weight. We are to find a solution-structure with minimal weight where by the *weight of a process graph* we mean the sum of the weights of the operating units belonging to the process graph under consideration. Now, an optimization problem, called PNS *problem*, can be formalized in the following way:

Let an instance $\mathbf{M} = (M, O, P, R)$ of the process design problem be given. Moreover, let w be a positive real-valued function defined on O, the *weight function*. The optimization problem is then

$$\min\{\sum_{u \in \bar{O}} w(u) : (\bar{M}, \bar{O}) \in S(M, O, P, R)\}. \tag{1}$$

It is worth noting that the PNS problem is NP-hard (*cf.* [1]).

As we mentioned some of the feasible solutions of (1), which are solution-structures, may represent non-executable processes. To illustrate this fact, let us consider the following simple example.

Example 1. Let $M = \{a_1, \ldots, a_6\}$, $R = \{a_1, a_2\}$, $P = \{a_6\}$ and $O = \{u_1, u_2, u_3\}$, where the definition of the operating units are given by the table below.

	input materials	output materials
u_1	a_1, a_2	a_3
u_2	a_3, a_4	a_5
u_3	a_5	a_4, a_6

Now, $S(M,O,P,R)$ is a singleton set and the only one solution-structure represent such a process which can not be executed. The corresponding process graph is depicted in Figure 1.

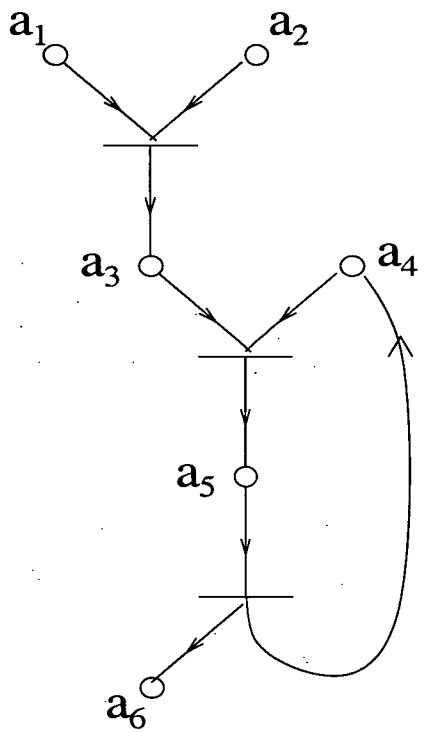


Figure 1. The process graph of Example 1.

For excluding the non-executability, we modify problem (1) in such a way that its feasible solutions will represent the executable feasible processes. For this reason, we use the following coloring of the process graphs. Let $(\bar{M},\bar{O})$ be a process graph and R a set of materials. It is said that $(\bar{M},\bar{O})$ is *colorable* by R if every material vertex of $(\bar{M},\bar{O})$ can be colored by the procedure below.

Coloring Procedure

Step 1. Color all of the materials in $\bar{M} \cap R$.

Step 2. If there is an operating unit whose all input materials have already colored, then color its all output materials. Terminate otherwise.

Now, we can define the modified optimization problem. Let a process design problem $\mathbf{M} = (M,O,P,R)$ be given. A subgraph $(\bar{M},\bar{O})$ of (M,O) is called a *feasible solution* of $(M,O,P,R,)$ if the following conditions are satisfied:

$(\bar{M}, \bar{O})$ satisfies $(A1)$ through $(A4)$,

moreover,

$(\bar{M}, \bar{O})$ is colorable by R.

The set of the feasible solutions of $\mathbf{M}$ is denoted by $S'(M, O, P, R,)$ or $S'(\mathbf{M})$. Let w be a positive real-valued function defined on O. Then the modified optimization problem can be defined as follows.

$$\min\{\sum_{u \in \bar{O}} w(u) : (\bar{M}, \bar{O}) \in S'(M, O, P, R)\}. \tag{2}$$

Let us investigate the relationship between the feasible solutions of (2) and the executable feasible processes. First let us consider an executable feasible process. Obviously it determines a subgraph $(\bar{M}, \bar{O})$ of (M, O) uniquely. The following properties can be excepted from an executable feasible process.

Evidently, it must be executable. This yields that $(\bar{M}, \bar{O})$ is colorable by R.

It has to produce every desired products. This results in $P \subseteq \bar{M}$, *i.e.* $(\bar{M}, \bar{O})$ satisfies $(A1)$.

A material can be regarded as a raw material in the process if it is not to be produced by any available operating unit of the process under consideration. On the other hand, it can be excepted that all the materials other than the raw materials are to be produced by some operating unit of the process. This implies that $(A2)$ is valid for $(\bar{M}, \bar{O})$.

The appearance of an operating unit in the structurally feasible process is forbidden unless the corresponding operating unit participates directly or indirectly in the production of the desired products. This yields that $(A3)$ holds for $(\bar{M}, \bar{O})$.

Each of the materials of the process must be consumed or produced by at least one of the operating units of the process. This implies that $(\bar{M}.\bar{O})$ satisfies $(A4)$.

Summarizing we have that the P-graph $(\bar{M}, \bar{O})$, determined by the executable feasible process considered, satisfies conditions $(A1)$ through $(A4)$, moreover, it is colorable by R, and hence it is a feasible solution of (2).

Now, let us consider the reverse situation. Let $(\bar{M}, \bar{O}) \in S'(\mathbf{M})$. Let us consider the process based on $(\bar{M}, \bar{O})$ from structural point of view. Such a process exists and unique. Since $(\bar{M}, \bar{O})$ is a subgraph of (M, O), the process consists of only available operating units. Condition $(A1)$ ensures that all the desired products are produced in this process. Condition $(A2)$ guarantees that all the unproduced materials available in the process are raw materials. Conditions $(A3)$ and $(A4)$ imply that this process does not contain unnecessary operating units and unnecessary materials. Finally, the colorability of $(\bar{M}, \bar{O})$ provides that the process is executable. Indeed, since the process graph is colorable, we can assign the time to every operating unit when its output materials are colored. Then by choosing a suitable time unit, the coloring time of every operating unit can be considered as its scheduling time, and the process is executable. Of course this scheduling is not necessarily optimal. Therefore, this process is an executable feasible process.

Obviously, problem (2) is such a restriction of (1), where we are to find a minimum-weight feasible solution among the feasible solutions of (1) which represent executable processes. We note that problem (2) is also NP-hard. It can be proved in the same way as for (1) (*cf.* [1]). We also note that if the process graph (M, O) of $\mathbf{M} = (M, O, P, R)$ is cycle-free, then $S'(\mathbf{M}) = S(\mathbf{M})$, and therefore, Problems (1) and (2) collapse. Regarding the solution of cycle-free PNS problems, we refer to [2], [3], and [9].

To close this section we recall some notions on automata. By an *automaton* we mean a system $\mathbf{A} = (A, X)$, where A is a finite nonvoid set of *states*, X is a finite nonempty set of *input signs*, and every $x \in X$ is realized as a unary operation $x^{\mathbf{A}}$ on A. For any $a \in A$ and $x \in X$, $ax^{\mathbf{A}}$ can be interpreted as the state into which $\mathbf{A}$ enters from a by receiving the input sign x. For a word $p \in X^*$, $ap^{\mathbf{A}}$ can be defined inductively as follows:

(1) $a\varepsilon^{\mathbf{A}} = a$,

(2) $ap^{\mathbf{A}} = (av^{\mathbf{A}})x^{\mathbf{A}}$ for $p = vx$, $v \in X^*$ and $x \in X$,

where ε denotes the empty word of X^*.

One can assign a directed *transition graph* to each automaton as follows. Let $\mathbf{A} = (A, X)$ be an arbitrary automaton. By the *transition graph* of $\mathbf{A}$ we mean the graph $\mathcal{G}_{\mathbf{A}} = (A, E)$, where for any couple of states $a, b \in A$, $(a, b) \in E$ if and only if there exists an input sign $x \in X$ such that $ax^{\mathbf{A}} = b$. Let us equip each edge of the transition graph with a label which is equal to the corresponding letter as usual.

A *recognizer* is a system $\mathcal{A} = (\mathbf{A}, a_0, F)$ which consists of an automaton $\mathbf{A} = (A, X)$, the *initial state* $a_0 (\in A)$, and the set $F (\subseteq A)$ of *final states*. The language *recognized* by $\mathcal{A}$ is

$$L(\mathcal{A}) = \{p : p \in X^* \text{ and } a_0 p^{\mathbf{A}} \in F\}.$$

It is also said that $L(\mathcal{A})$ is *recognizable* by the automaton $\mathbf{A}$.

2 Solution of the modified PNS problem

We shall solve problem (2), by using an automaton theoretical approach. Namely, for every instance of (2), an automaton is constructed such that some feasible solutions of (2) can be described as words over the input alphabet of this automaton, moreover, these words are accepted by a recognizer based on this automaton. Then, by equipping the transition graph of the automaton considered with the weigths of the operating units, a shortest path in this weighted graph which leads from the initial state into the set of final states of this recognizer provides an optimal solution of (2).

We shall use the following statement.

Lemma. *Let an instance* $\mathbf{M} = (M, O, P, R)$ *of the process design problem be given. Moreover, let* $(\bar{M}, \bar{O})$ *be a process graph which is colorable by* R *and satisfies conditions* $(A1), (A2)$, *and* $(A4)$, *but* $(A3)$ *is not valid for* $(\bar{M}, \bar{O})$. *Then there exists a proper subgraph of* $(\bar{M}, \bar{O})$ *which is colorable by* R *and satisfies all the four conditions.*

Proof. We present a procedure to construct the required subgraph.

Procedure

Initialization Let $K_0 = P$, $O_0 = \emptyset$, and $i = 0$

Iteration (i-*th* iteration)

Step 1. Terminate if $K_i \subseteq R$; the required subgraph is $(mat(O_i), O_i)$. Otherwise proceed to Step 2.

Step 2. Select a material $x \in K_i \setminus R$ and an operating unit $u \in \bar{O}$ such that $x \in mat^{out}(u)$. Let $O_{i+1} = O_i \cup \{u\}$ and $K_{i+1} = (K_i \cup mat^{in}(u)) \setminus mat^{out}(O_{i+1})$. Set $i := i + 1$, and proceed to the succeeding iteration step.

The procedure is correct, since the colorability of $(\bar{M}, \bar{O})$ implies that if $K \not\subseteq R$ $(K \subseteq \bar{M})$, then there are $x \in K \setminus R$ and $u \in \bar{O}$ with $x \in mat^{out}(u)$. Now, let us suppose that the procedure is finished by the process graph (M_i, O_i), where $M_i = mat(O_i)$. Obviously, (M_i, O_i) is a subgraph of $(\bar{M}, \bar{O})$, moreover, $K_i \subseteq R$. These facts imply that (M_i, O_i) satisfies condition $(A2)$. From $M_i = mat(O_i)$ it follows that (M_i, O_i) satisfies $(A4)$. $K_0 = P$ implies that $(A1)$ is valid for (M_i, O_i). Finally, from the procedure it follows that $(A3)$ is also valid for (M_i, O_i), and therefore, (M_i, O_i) satisfies all the four conditions. Now, if (M_i, O_i) is not a proper subgraph of $(\bar{M}, \bar{O})$, then the two process graphs are equal. But this equality contradicts our assumption that $(\bar{M}, \bar{O})$ does not satisfy condition $(A3)$. Consequently, (M_i, O_i) is a proper subgraph of $(\bar{M}, \bar{O})$. Finally, it can be proved by induction on j that if each material of K_j has got color, then the materials contained in $mat(O_j)$ can be colored by K_j. From this fact it follows that (M_i, O_i) can be colored by R, which ends the proof of the statement.

To construct the automaton mentioned above, let us consider an arbitrary instance $\mathbf{M} = (M, O, P, R)$ of the design problem and let w be a weight function.

Let us define the automaton $\mathbf{B} = (B, O')$ as follows. Let $B = B' \cup \{\diamond\}$, where $B' = \wp'(M)$ and $\diamond \notin B'$. Moreover, let $O' = \{u : u = (C, D) \in O \text{ and } R \cap D = \emptyset\}$. One can give the states of the automaton the following meaning. A state which is a set of materials means the available materials at a given time. State $\diamond$ is used for describing the unsuccessful transitions. The transitions are defined in the following way. For every $Q \in B'$ and $u = (C, D) \in O'$, let

$$Qu^{\mathbf{B}} = \begin{cases} Q \cup D & \text{if } C \subseteq Q, \\ \diamond & \text{otherwise,} \end{cases} \qquad \text{moreover, let } \diamond u^{\mathbf{B}} = \diamond.$$

The transitions have the following meaning. Let us suppose that we are going to build up a process graph. First we fix the available materials, their set Q will be the starting state of the automaton. As a next step, we try to put an operating unit in the graph, let $u = (C, D)$ denote it. If each input material of u is available at this moment, *i.e.*, $C \subseteq Q$, then we can put u in the process graph, and we can suppose that from this moment the available materials are the earlier available ones and the output materials of u, *i.e.*, the elements of the set $Q \cup D$. If u has such an input material which is not available, then we can not put u in the process graph we build, and this fact is expressed such that the transition is unsuccessful. It is easy to check that the following observation is valid for the automaton $\mathbf{B}$.

Remark 2. If Q is a state of $\mathbf{B}$, p is a word over O', and $u \in O'$ occurs in p, then $Q(pu)^{\mathbf{B}} = Qp^{\mathbf{B}}$.

Let us equip the transition graph $\mathcal{G}_{\mathbf{B}}$ with weights in the following way. If (Q, Q') is an edge of $\mathcal{G}$ and the labels of this edge are $u_{j_1}, \ldots, u_{j_t}$, then let us assign the weight $w' = \min\{w(u_{j_1}), \ldots, w(u_{j_t})\}$ to the edge under consideration, moreover, if $w' = u_{j_l}$ for some $1 \leq l \leq t$, then keep the label u_{j_l} and cancel the remaining labels of this edge. Let us denote this weighted and labelled graph by $(\mathcal{G}_{\mathbf{B}}, w)$.

Let us define now the recognizer $\mathcal{B} = (\mathbf{B}, R, F)$, where $F = \{Q : Q \subseteq B' \text{ and } P \subseteq Q\}$. Then the following statement is valid.

Proposition. *For every word $p = u_{i_1} \ldots u_{i_k} \in L(\mathcal{B})$, if $path[R, Rp^{\mathbf{B}}]$ is a shortest path among the paths leading from R into a final state in $(\mathcal{G}_{\mathbf{B}}, w)$, then $u_{i_1}, \ldots, u_{i_k}$ are pairwise different and $(\bar{M}, \bar{O})$ is an optimal solution of (2), where $\bar{O} = \{u_{i_1}, \ldots, u_{i_k}\}$ and $\bar{M} = mat(\bar{O})$.*

Proof. Let $p = u_{i_1} \ldots u_{i_k} \in L(\mathcal{B})$ and let us suppose that $path[R, Rp^{\mathbf{B}}]$ is a shortest path leading from R into a final state in $(\mathcal{G}_{\mathbf{B}}, w)$. Then Remark 2 implies that $u_{i_1}, \ldots, u_{i_k}$ are pairwise different, since every operating unit has a positive weight. Now, let us consider the process graph $(\bar{M}, \bar{O})$, where $\bar{O} = \{u_{i_1}, \ldots, u_{i_k}\}$ and $\bar{M} = mat(\bar{O})$. First we show that $(\bar{M}, \bar{O})$ is a feasible solution of (2). From the definition of $(\bar{M}, \bar{O})$ it follows that $(A4)$ holds for $(\bar{M}, \bar{O})$. The definition of O' and $p \in L(\mathcal{B})$ imply that $(A2)$ is valid for $(\bar{M}, \bar{O})$. Moreover, from $p \in L(\mathcal{B})$ it follows that $(\bar{M}, \bar{O})$ is colorable by R and $(A1)$ is valid for $(\bar{M}, \bar{O})$. It is stated now that $(\bar{M}, \bar{O})$ satisfies $(A3)$. If it is not so, then by our Lemma, there exists a proper subgraph of $(\bar{M}, \bar{O})$ which is a feasible solution of (2). Let us denote this subgraph by $(\widehat{M}, \widehat{O})$. Since $\bar{M} = mat(\bar{O})$ and by Remark 1, $\widehat{M} = mat(\widehat{O})$, we obtain that $\widehat{O} \subset \bar{O}$. Let us suppose that $\widehat{O} = \{u_{j_1}, \ldots, u_{j_l}\} \subset \bar{O}$ for some $1 \leq l < k$. Since $(\widehat{M}, \widehat{O})$ is a feasible solution of (2), it is colorable by R. Without loss of generality, we may assume that the coloring procedure first colors the output materials of u_{j_1}, then the output materials of u_{j_2} *etc.* This yields that the word $\hat{p} = u_{j_1} \ldots u_{j_l}$ brings the automaton $\mathbf{B}$ from R into some final state. On the other hand, the weight of $path[R, R\hat{p}^{\mathbf{B}}]$ is less than the weight of $path[R, Rp^{\mathbf{B}}]$ which is a

contradiction. Therefore, $(\bar{M}, \bar{O})$ satisfies $(A3)$, and it is a feasible solution of (2). Let us observe that the weight $\bar{w}$ of $(\bar{M}, \bar{O})$ is equal to the weight of $path[R, Rp^{\mathbf{B}}]$. Now, we prove that $(\bar{M}, \bar{O})$ is an optimal solution of (2). Indeed, if it is not so, then there exists a feasible solution $(\widehat{M}, \widehat{O})$ of (2) such that its weight $\hat{w}$ is less than the weight $\bar{w}$ of $(\bar{M}, \bar{O})$. In similar way as above, we can construct then a word $\hat{p}$ such that $\hat{p} \in L(\mathbf{B})$ and $\hat{w}$ is equal to the weight of $path[R, R\hat{p}^{\mathbf{B}}]$. This yields that the weight of $path[R, R\hat{p}^{\mathbf{B}}]$ is less than the weight of $path[R, Rp^{\mathbf{B}}]$ which contradicts our assumption that $path[R, Rp^{\mathbf{B}}]$ is a shortest path leading from R into some final state in $(\mathcal{G}_{\mathbf{B}}, w)$. Consequently, $(\bar{M}, \bar{O})$ is an optimal solution of (2).

Our Proposition provides the following procedure for finding an optimal solution of (2).

Procedure for finding an optimal solution of (2)

Step 1. Construct the transition graph of the automaton $\mathbf{B}$ and calculate the set F of final states.

Step 2. Let us equip the transition graph with the weights of the operating units, and simultaneously, rewrite the labels of the edges such that let every edge have only one label.

Step 3. Determine a shortest path leading from the state R into the set F.

Step 4. By using the obtained shortest path, determine an optimal solution of problem (2).

It is worth noting that the whole transition graph is not required by the procedure in general, only the transition graph of the subautomaton generated by the state R. To demonstrate this fact and the procedure, let us consider the following small example.

Example 2. Let $M = \{a_1, \ldots, a_8\}$, $R = \{a_1, a_2, a_3\}$, $P = \{a_8\}$ and $O = \{u_1, u_2, u_3, u_4\}$, where the definition of the operating units and their weights are given by the table below.

	input materials	output materials	weight
u_1	a_1, a_2	a_4, a_5	2
u_2	a_2	a_5, a_6	5
u_3	a_1, a_3	a_6, a_7	1
u_4	a_5, a_6	a_8	3

The process graph of this design problem is depicted in Figure 2.

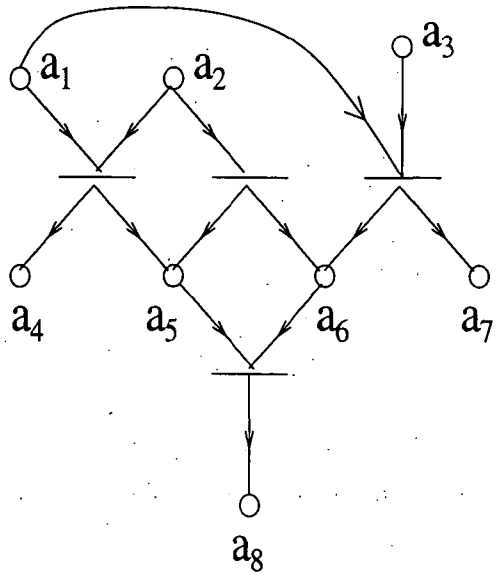


Figure 2. The process graph of Example 2.

By constructing the transition graph of the subautomaton generated by R, we get a transition graph of 12 vertices. It is depicted in Figure 3, where the sets are given by circles containing the indices of their elements, loop edges are omitted, furthermore, over each edge the operating unit is written which induces the transition and under the edge the weight of the operating unit is given. By determining the shortest paths, we obtain that the path belonging to the word $u_1u_3u_4$ is a suitable shortest path, its edges are bold in Figure 3. The corresponding optimal solution with weight 6 is given in Figure 4.

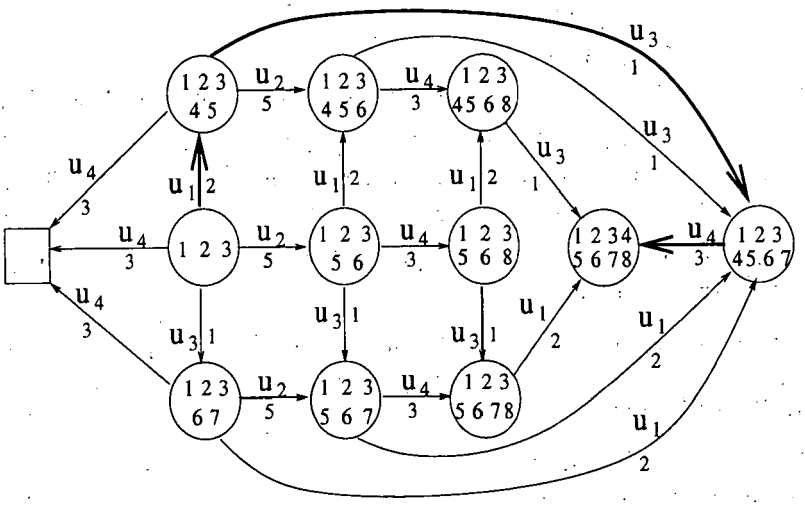


Figure 3. The weighted transition graph for Example 2.

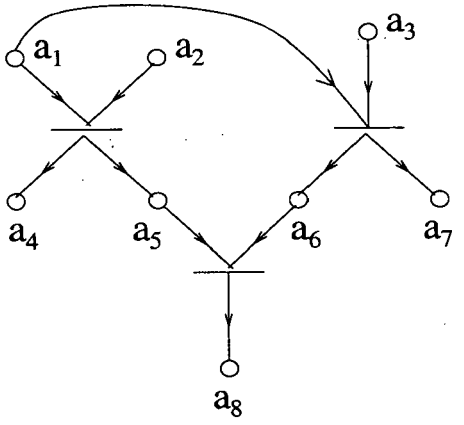


Figure 4. The optimal solution of Example 2.

References

[1] Blázsik, Z. and B. Imreh, A note on connection between PNS and set covering problems, *Acta Cybernetica* **12** (1996), 309-312.

[2] Blázsik, Z., Cs. Holló, B. Imreh, Cs. Imreh, Z. Kovács, On a well-solvable class of the PNS problem, *Novi Sad Journal of Mathematics* **30** (2000), 21-30.

[3] Blázsik, Z., Cs. Holló, Cs. Imreh, Z. Kovács, Heuristics for the PNS Problem, *Mátraháza Optimization Days*, editors: F. Gianessi, P. Pardalos, T. Rapcsák, Kluwer Academic Publisher, to appear.

[4] Friedler, F., L. T. Fan, B. Imreh, Process Network Synthesis: Problem Definition, *Networks* **28** (1998), 119-124.

[5] Friedler, F., K. Tarján, Y. W. Huang, and L. T. Fan, Graph-Theoretic Approach to Process Synthesis: Axioms and Theorems, *Chem. Eng. Sci.* **47**(8) (1992), 1973-1988.

[6] Friedler, F., K. Tarján, Y.W. Huang, and L.T. Fan, Combinatorial Algorithms for Process Synthesis, *Computers Chem. Engng.* **16** (1992), S313-S320.

[7] Imreh, B., F. Friedler, L. T. Fan, An Algorithm for Improving the Bounding Procedure in Solving Process Network Synthesis by a Branch-and-Bound Method, *Developments in Global Optimization*, (editors: I. M. Bonze, T. Csendes, R. Horst, P. M. Pardalos), Kluwer Academic Publisher, Dordrecht, Boston, London, 1996, 301-348.

[8] Imreh, B., Z. Kovács, A note on separation-networks and automata, *Publicationes Mathematicae*, submitted for publication.

[9] Imreh, Cs., A new well-solvable class of PNS problems, *Computing* **66** (2001), 289-296.

Received March, 2001

Acta Cybernetica **15** (2002) 339–351.

Factorizations of Languages and Commutativity Conditions *

Alexandru Mateescu[†], Arto Salomaa[‡], and Sheng Yu[§]

Abstract

Representations of languages as a product (catenation) of languages are investigated, where the factor languages are "prime", that is, cannot be decomposed further in a nontrivial manner. In general, such prime decompositions do not necessarily exist. If they exist, they are not necessarily unique - the number of factors can vary even exponentially. The paper investigates prime decompositions, as well as the commuting of the factors, especially for the case of finite languages. In particular, a technique about commuting is developed in Section 4, where the factorization of languages L_1 and L_2 is discussed under the assumption $L_1L_2 = L_2L_1$.

Keywords: finite language, catenation, commutativity of languages, prime decomposition

1 Introduction

Prime factorizations of natural numbers and their uniqueness constitute one of the really fundamental issues in all mathematical sciences. On the other hand, in the theory of formal languages, the operation of *product* or *catenation* was introduced already at a very early stage. Clearly, any language L can be expressed as a product of itself and the language $\{\lambda\}$ consisting of the empty word λ. We refer to such decompositions of L as *trivial*, and say that $L \neq \{\lambda\}$ is *prime* if it has only trivial decompositions. In a *prime decomposition* for a language L every factor is a prime. Although questions dealing with primality can be viewed as fundamental in language theory, rather little work in this area has been done so far, see, for instance, [10, 6]. [2] is an early reference dealing with finite languages. [7] develops

*This work has been partially supported by the Project 137358 of the Academy of Finland and by the Natural Sciences and Engineering Research Council of Canada grants OGP0041630. All correspondence to Sheng Yu.

[†]Faculty of Mathematics, University of Bucharest Academiei, 14, Bucharest, Romania E-mail: alexmate@pcnet.pcnet.ro

[‡]Turku Centre for Computer Science (TUCS) Lemminkäisenkatu 14A, 20520 Turku, Finland E-mail: asalomaa@utu.fi

[§]Department of Computer Science, University of Western Ontario London, Ontario, Canada N6A 5B7 E-mail: syu@csd.uwo.ca

a method according to which one may construct, with the maximal use of the distributive law, for every finite language F an expression from which the number of states and final states in the minimal deterministic automaton for F can be immediately seen. [10] contains results about the commuting of two languages in some special cases.

The following remarks about related papers are in order. A systematic study about decompositions was initiated in the technical report [5]. This paper is the "journal version" of the report [5], while [9] is the "conference version" of it. The report [5] has given impetus to further research, for instance, [3, 4]. We have included in this paper material from [9] only insofar it increases readability. In particular, our main technical contribution in this paper, Section 4, is disjoint from [9].

We begin with the following basic observation about finite languages. Whenever a nonempty finite language F can be written as a product

$$F = F_1 F_2 \dots F_k,$$

where none of the factors F_i, $1 \leq i \leq k$, is trivial, then k cannot be larger than the length of the longest word in F. Consequently, we have always a complete control of all possible decompositions, at least in principle. This does not hold true for infinite regular languages, where there is no bound for the number of factors. Still decompositions such as

$$\Sigma^* = L_1 L_2 = (\lambda + \Sigma + \Sigma^2 + \dots \Sigma^{n-1})(\Sigma^n)^*, n \geq 2,$$

convey definite information about Σ^*. (Here, as frequently in the sequel, "+" stands for union.) Indeed, they were instrumental in the proof for the fact that equations between regular expressions possess no finite basis, see [7] for details.

Every finite language (different from $\{\lambda\}$) possesses a prime decomposition. This follows by an obvious induction on the length of the longest word in the language. This is not true for infinite languages. For instance, no star language L ($L = K^*$, for some K) can possess a prime decomposition. Indeed, for infinite languages, decompositions other than prime decompositions can sometimes be quite useful. For instance consider the language L over the one-letter alphabet $\{a\}$,

$$L = \{a^i \mid i = 10, 13, 16, 17, 19, 20 \text{ or } i \geq 22\}.$$

L possesses a decomposition $L = L_1 L_2$, where $L_1 = (a^3)^+$ and $L_2 = (a^7)^+$. Here we definitely have a simplification of the original language, presented as a product of languages, although the factors are not prime. For instance, the total number of states in the minimal automata for L_1 and L_2 is much smaller than the number of states in the minimal automaton of L. Using the same idea and allowing an arbitrary number of factors, one can show that the number of states may grow exponentially in the transition from the decomposition to the original language. Somewhat similar matters are discussed also in Section 3. We hope to return to the discussion of this and other similar problems (which lie outside the scope of the present paper) in another context.

A brief description of the contents of the present paper follows. The reader is expected to be familiar with the very basics of formal languages and finite automata. One of the references [6, 7, 8] may be consulted if need arises.

Basic decidability results are presented in Section 2. They lead also to a notion very central in the study of regular languages, that of a *decomposition set*, originally introduced in [9]. Sections 3-5 deal exclusively with finite languages. In Section 3, we discuss decompositions of different lengths, as well as the testing of the primality of a finite language, also from the point of view of complexity. The two final sections deal with the *commuting* of two finite languages F_1 and F_2, that is, the validity of the equation $F_1F_2 = F_2F_1$. While this is a tricky problem in the general case, some special cases can be handled.

Our main results are contained in Section 4, where factorization of languages F_1 and F_2 is discussed under the assumption $F_1F_2 = F_2F_1$. Also a very efficient construction is presented in the case where one of the two languages involved is a singleton. The construction could be applicable also in other similar situations. The final Section 5 discusses some recent results and open problems.

It has not escaped our notice, especially in view of the many possible interpretations of finite languages and the central theoretical role of the problems studied in this paper, that the problems might turn out to be significant in certain applications. For instance, succinct representations of DNA nucleotide sequences certainly fall within this category. However, we have had no specific applications in mind.

2 Decomposition sets and decision problems

The notions of a *prime language* and a *prime decomposition* of a language were already defined in the Introduction. According to the definition, the language $\{\lambda\}$ consisting of the empty word is not prime. Thus, all factor languages in a prime decomposition are nontrivial. Depending on the language, the prime decomposition may be unique or there may be several prime decompositions for the same language. It is also possible that a language has no prime decompositions. However, every finite language possesses a prime decomposition.

Typical problems concerning the decomposition of finite languages are the following:

1. Is a given finite language prime?

2. Find all prime decompositions of a given finite language.

3. Find, for a given finite language, a prime decomposition possessing a specific property. (We might require, for instance, that the total number of states in the automata accepting the prime factors is minimal.)

It is obvious that all problems of this nature are decidable for finite languages. The complexity issues lie mainly outside the scope of this paper. In many cases, an exhaustive search is the only algorithm we know for a specific problem.

We now present some simple examples, due to [9], of prime and nonprime finite langauges. Consider the languages over the alphabet $\{a\}$, defined by

$$F_{n,k} = \lambda + a^k + a^{2k} + \cdots + a^{nk}, n \geq 2, k \geq 1,$$

$$F'_n = \lambda + a^2 + a^3 + a^4 + \cdots + a^n, n \geq 4.$$

Let, further, $F''_n, n \geq 4$, denote any language consisting of λ and a^n and, in addition, of arbitrarily many words a^i with $n/2 \leq i \leq n$. Then no language $F_{n,k}$ is prime, whereas all languages F''_n are prime. The language F'_n is prime iff $n = 4$.

Sometimes a slight change in a prime language induces a possibility for a decomposition. Consider the following two languages:

$$F = adba + acbb + bcaa + bdab,$$

$$F' = adba + adbb + bdaa + bdab.$$

Thus F' results from F by replacing the two occurrences of the letter c by the letter d. Then the language F is prime, whereas F' possesses the decomposition $F' = (adb + bda)(a + b)$. See [9] for details, as well as for the proof of the following theorem and for related references.

Theorem 1 *There is no algorithm for deciding whether or not a given linear language is prime. Consequently, the problem of primality is undecidable for context-free languages.*

The proof of Theorem 1 does not work for regular languages. Indeed, as Theorem 2.2 below shows, the primality problem is decidable for regular languages. We now recall from [9] a notion very suitable for the study of decompositions of regular languages. It is closely related to left quotients of regular languages. It shows how an arbitrary decomposition can be extended to one of finitely many specific decompositions, obtainable in a standard way.

Let R be a regular language over an alphabet Σ, and let $A = (Q, \Sigma, \delta, q_0, Q_F)$ be the minimal finite deterministic automaton for R. (Here Q is the set of states, q_0 the initial state, Q_F the set of final states, and δ the transition function. We extend δ to words over Σ. Thus, $\delta(q, w) = q'$ means that the word w takes A from the state q to the state q'.) For a nonempty subset $P \subseteq Q$, we consider the following two languages:

$$R_1^P = \{w \mid \delta(q_0, w) \in P\},$$

$$R_2^P = \bigcap_{p \in P} \{w \mid \delta(p, w) \in Q_F\}.$$

Lemma 1 *Let R and A be defined as above. Assume that $R = L_1 L_2$, where L_1 and L_2 are arbitrary languages. Define $P \subseteq Q$ by*

$$P = \{p \in Q \mid \delta(q_0, w) = p, \text{ for some } w \in L_1\}.$$

Then $R = R_1^P R_2^P$ and, moreover, $L_i \subseteq R_i^P$ for $i = 1, 2$.

Lemma 1 was established in [9]. Observe that the languages L_1 and L_2 above are quite arbitrary; they need not even be recursively enumerable. They can always be extended, without losing the validity of the decomposition, to regular languages obtainable from the minimal automaton for A. These resulting "standard" decompositions can always be expressed in terms of a *decomposition set.*

By definition, a nonempty subset $P \subseteq Q$ is a *decomposition set* (for a regular language R) if $R = R_1^P R_2^P$. The decomposition $R = R_1^P R_2^P$ referred to as the decomposition of R *induced* by the decomposition set P. We say that the decomposition $L = L_1 L_2$ of a language L is *included* in the decomposition $L = L_1' L_2'$ if $L_i \subseteq L_i'$, $i = 1, 2$. See [9] for the proof of the following result.

Theorem 2 *Every decomposition of a regular language R is included in a decomposition of R induced by a decomposition set. The problem of primality is decidable for regular languages.*

The algorithm obtained by checking through all possible decomposition sets is clearly exponential. It is likely that primality testing is NP-complete even for finite languages. Observe also that the decomposition induced by a decomposition set may be trivial. Indeed, we have $R_1^P = \{\lambda\}$ iff $P = \{q_0\}$ and q_0 has no incoming arrows. Similarly, $R_2^P = \{\lambda\}$ exactly in case $P = Q_F$ and λ is the only word taking A from each of the final states to a final state. Also the following result is an immediate corollary of Lemma 1.

Theorem 3 *Whenever a regular language has a nontrivial decomposition, it has a nontrivial decomposition where the factors are regular languages.*

We conclude this section with two open problems.

Open problem. Instead of catenation, we may take the *shuffle* operation to be the *product* operation for languages. Decompositions and primality can be defined for this product as well. Is the last sentence of Theorem 2 valid also now? In other words, is the primality of regular languages with respect to the shuffle product decidable? Although we have been able to settle some special cases, the case of an arbitrary regular language seems to be very tricky.

Open problem. Does Theorem 3 hold with "regular" replaced by "context-free"? It would be very strange to have an example of a context-free language L having nontrivial decompositions $L = L_1 L_2$, in all of which at least one of the languages L_1 and L_2 is non-context-free.

3 Primality testing

In the remainder of this paper we discuss only finite languages. A given finite language may possess several prime decompositions. It may even happen that two prime decompositions of the same language have no common factors. For instance,

$$(\lambda + a^2)(\lambda + a^2 + a^3 + a^4) = (\lambda + a^2 + a^3)^2,$$

where all factors are prime languages. Even the number of factors may vary drastically in different prime decompositions of the same language. The following contribution to Problem 2 of the preceding section was established in [9].

Theorem 4 *There are finite languages L_n having two prime decompositions with $O(n)$ and $O(\log n)$ factors.*

Theorem 4 was established in [9] using the following example. Consider numbers $n = 2^k$, $k \geq 1$, and languages

$$L_n = \lambda + a + a^2 + \cdots + a^{n-1}.$$

Then

$$L_n = (\lambda + a)^{n-1} = (\lambda + a)(\lambda + a^2)(\lambda + a^4)\ldots(\lambda + a^{2^{k-1}}).$$

The most straightforward examples about factorizations not unique are obtained in terms of languages over one-letter alphabet $\{a\}$. Other examples are easy to construct. For instance,

$$F' = \lambda + a + b + ab + b^2 + ab^2 + b^3 + ab^3 + b^4 + ab^4 = (\lambda + a + b + b^2 + ab^2)(\lambda + b)^2 =$$
$$= (\lambda + a + ab + b^2 + ab^2)(\lambda + b)^2 = (\lambda + a)(\lambda + b^2)(\lambda + b)^2,$$

where all languages within parentheses are primes.

Consider primality testing, Problem 1 mentioned in Section 2. There seems to be no other general method than trying all possible factors. Of course, in special instances, ad hoc arguments can be used to exclude factors of certain types. A special case consists of testing the primality of languages of the form

$$\lambda + a^{i_1} + a^{i_2} + \cdots + a^{i_n}, \tag{1}$$

where the i's are distinct positive integers. In this case primality testing can be reduced to a problem concerning sets of nonnegative integers as follows.

Let N be a set of nonnegative integers. We say that N has the *decomposition property* if there are nonempty subsets N_1 and N_2 of N, maybe overlapping or identical but both containing at least two elements, such that

$$N = \{n_1 + n_2 \mid n_1 \in N_1 \text{ and } n_2 \in N_2\}.$$

We also say that N *decomposes into* N_1 and N_2. (Recall here also the one-letter language L presented in the Introduction.)

Clearly, N can have the decomposition property only if $0 \in N$, in which case 0 belongs also to both N_1 and N_2. The following result is now obvious.

Lemma 2 *The language $L_N = \Sigma_{i \in N} a^i$ is prime iff the set N contains 0 and has not the decomposition property. More specifically, if N decomposes into N_1 and N_2 then*

$$L_N = (\lambda + \sum_{i \in N_1} a^i)(\lambda + \sum_{i \in N_2} a^i)$$

Although the problem of N possessing the decomposition property bears some resemblance to the subset sum problems, we have not been able to establish its NP-completeness. Of course, testing the primality of the languages (1) is only a special case of the general problem.

If c is a letter not in the alphabet of F, then $F + c$ is always prime. One can affect the same change also without introducing new letters.

Theorem 5 *Let F be a finite language whose minimal alphabet Σ contains at least two letters. Then for some $w \in \Sigma^+$, $F + w$ is prime.*

Proof. Let k be the length of the longest word in F. Let w be any word of length $2k+1$ such that there is a word in F whose first (resp. last) letter differs from the first (resp. last) letter of w. This requirement can be satisfied since Σ contains at least two letters. We claim that $F + w$ is prime. Assume the contrary: $F + w$ has a nontrivial decomposition $F + w = F_1F_2$. We can write $F_1 = F_1' + w_1$, $F_2 = F_2' + w_2$, $w = w_1w_2$. (Possibly F_i' is empty or $w_i = \lambda$.) One of the words w_1 and w_2 is of length greater than k. Assume that $|w_2| > k$. Then $F_1' = \emptyset$ because, if $x \in F_1'$, the word xw_2 is not in $F + w$. Thus, $F + w = w_1F_2$. But this is not possible because F contains a word whose first letter differs from the first letter of w_1. ($w_1 = \lambda$ would yield a trivial decomposition.) If $|w_1| > k$, we obtain similarly a contradiction, using the fact that F contains a word whose last letter differs from the last letter of w_2. This completes the proof. □

Theorem 5 can be extended to concern languages F over $\{a\}$ containing the empty word.

4 Factorization versus commutativity conditions

It was one of the very early results on combinatorics on words that two words u and v *commute*, $uv = vu$, iff both u and v are powers of the same word. No similar result is known for finite languages. When do two finite languages F_1 and F_2 commute, $F_1F_2 = F_2F_1$? We begin with the special case, where one of the languages is a singleton. The technique presented in this section, interesting also on its own right, shows in detail the structure of the two languages.

The following results are well known and can be found in, e.g., [6] or [10].

Lemma 3 *If $uv = vz$, $u, v, z \in \Sigma^*$, and $u \neq \lambda$, then $u = xy$, $v = (xy)^k x$, and $z = yx$ for some $x, y \in \Sigma^*$ and $k \geq 0$.*

Lemma 4 *If $uv = vu$, then there exists $x \in \Sigma^*$ such that $u = x^s$ and $v = x^t$ for some $s, t \geq 0$.*

Lemma 5 *If $u^m = v^n$ and $m, n \geq 1$, then $u = x^s$ and $v = x^t$ for some $x \in \Sigma^*$ and $s, t \geq 1$.*

Theorem 6 *Let $x \in \Sigma^*$ and $L \subseteq \Sigma^*$ be a finite language. If $xL = Lx$, then there exists $w \in \Sigma^*$ such that $x = w^s$ and $L = \bigcup_{i=1}^{n} \{w^{t_i}\}$, for $s, n, t_1, \ldots, t_n \geq 0$.*

Proof. The theorem holds trivially if $L = \emptyset$. If $L = \{y\}$, then $xy = yx$. By Lemma 4, we have $x = w^s$ and $y = w^t$ for some $s, t \geq 0$. Thus, the theorem holds.

Assume that the theorem holds for $L = \{y_1, \ldots, y_t\}$, $t < n$.

Now we consider the case when $t = n$, i.e., $L = \{y_1, \ldots, y_n\}$. We have the following three cases:

Case I. $xy_n = y_n x$. Then, by Lemma 4, $x = w_0^{s_0}$ and $y_n = w_0^{t_0}$ for some $w_0 \in \Sigma^*$ and $s_0, t_0 \geq 0$. Let $L' = \{y_1, \ldots, y_{n-1}\}$. Then $xL' = L'x$ since $xL = Lx$, $xy_n = y_n x$, and $xy_n \notin xL'$ and $y_n x \notin L'x$. By the induction hypothesis, $x = w_1^{s_1}$ and $L' = \bigcup_{i=1}^{n-1} \{w_1^{t_i}\}$ for some $w_1 \in \Sigma^*$ and $s_1, t_1 \geq 0$. Since $x = w_0^{s_0} = w_1^{s_1}$, w_0 and w_1 are powers of a common word w, i.e., $w_0 = w^l$ and $w_1 = w^m$. Then $x = w^{ls_0}$ and $L = \{w^{mt_1}, \ldots, w^{mt_{n-1}}, w^{lt_0}\}$. The theorem holds.

Case II. $xy_n \neq y_n x$. Then $xy_n = y_{i_1} x$ for some $i_1 \in \{1, \ldots, n-1\}$. If $xy_{i_1} = y_n x$, then let $L_1 = \{y_{i_1}, y_n\}$ and $L_2 = L - L_1$. Otherwise, $xy_{i_1} = y_{i_2} x$ for some $i_2 \in \{1, \ldots, n-1\} - \{i_1\}$. We continue this way until we get $xy_{i_m} = y_n x$, i.e.

$$xy_n = y_{i_1} x, \quad xy_{i_1} = y_{i_2} x, \quad \ldots\ldots, \quad xy_{i_m} = y_n x.$$

Consider the case $m < n - 1$. Let $L_1 = \{y_{i_1}, \ldots, y_{i_m}, y_n\}$ and $L_2 = L - L_1$. Then $xL_1 = L_1 x$ and $xL_2 = L_2 x$. By the induction hypothesis, we have

$$x = u^{s_1}, \quad L_1 = \bigcup_{i=1}^{m} \{u^{t_i}\}, \text{ and } x = v^{s_2}, \quad L_2 = \bigcup_{j=1}^{n} \{v^{t_j}\}.$$

Since $u^{s_1} = v^{s_2} = x$, we have $u = w^k$ and $v = w^l$ for $w \in \Sigma^*$ and $k, l \geq 0$. Therefore,

$$x = w^{ks_1}, \quad L = (\bigcup_{i=1}^{m} \{w^{kt_i}\}) \cup (\bigcup_{j=1}^{n} \{w^{lt_j}\}).$$

Case III. This case is the same as Case II except that $m = n - 1$, i.e, we have $xy_n \neq y_n x$ and

$$xy_n = y_{i_1} x, \quad xy_{i_1} = y_{i_2} x, \ldots\ldots, \quad xy_{i_{n-1}} = y_n x.$$

Since $xy_n = y_{i_1} x$ and $xy_{i_1} = y_{i_2} x$, we have, by Lemma 3,

$$x = (u_1 v_1)^{k_1} u_1, \quad y_n = v_1 u_1, \quad y_{i_1} = u_1 v_1$$

$$x = (u_2 v_2)^{k_2} u_2, \quad y_{i_1} = v_2 u_2, \quad y_{i_2} = u_2 v_2.$$

So, we have

$$(u_1 v_1)^{k_1} u_1 u_1 v_1 = u_2 v_2 (u_1 v_1)^{k_1} u_1.$$

Then, $u_1 v_1 = v_1 u_1$. Thus, u_1 and v_1 are powers of the same word $w_1 \in \Sigma^*$. So, $x = w_1^{s_1}$ and $y_1 = w_1^{t_1}$ for $s_1, t_1 \geq 0$. Similarly, we can show that, for $1 \leq i \leq n$,

$$x = w_i^{s_i} \text{ and } y_i = w_i^{t_i}.$$

Since $w_1^{s_1} = \ldots = w_n^{s_n} = x$, we know that $w_1, \ldots, w_n$ are powers of a common word w, i.e., $w_1 = w^{l_1}, \ldots, w_n = w^{l_n}$ by Lemma 5. Thus, $L = \bigcup_{i=1}^{n} \{w^{l_i t_i}\}$ and $x = w^{l_1 s_1}$. □

Let p and q be two natural numbers such that $(p,q) = 1$ and $p < q$. Define $N_p = \{1, \ldots, p\}$ and $N_q = \{1, \ldots, q\}$. Also define a function $\sigma : N_q \to N_q$ by $\sigma(i) = ((i+p-1) \bmod q)+1$. Thus, $\sigma(i) = i+p$ where the least positive remainder of the sum modulo q is taken. Since $(p,q) = 1$, it is clear that, for any $i \in N_q$, we have $\{i, \sigma(i), \ldots, \sigma^{q-1}(i)\} = N_q$ and $\sigma^q(i) = i$.

Let $w \in \Sigma^{tm}$, $t, m > 0$, i.e., $w = x_1 x_2 \cdots x_t$ and $x_i \in \Sigma^m$, $1 \le i \le t$. Denote by $(w)_i^{(m)}$, $1 \le i \le t$, the substring x_i of w. When m is understood, we simply write $(w)_i$.

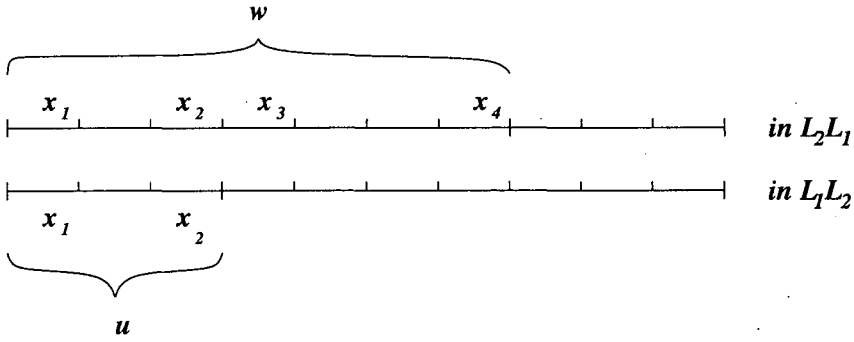
Let $L_1 \subseteq \Sigma^{pm}$, $L_2 \subseteq \Sigma^{qm}$, $p, q, m > 0$, $(p,q) = 1$, $p < q$, and $L_1 L_2 = L_2 L_1$. Then we have the following results.

Lemma 6 *Let $N_q' = \{i_1, \ldots, i_n\}$, $1 \le n \le q$, be a subset of N_q and $x_1, \ldots, x_n \in \Sigma^m$. If there exists $w \in L_2$ such that $(w)_{i_1} = x_1, \ldots, (w)_{i_n} = x_n$, then there exists $u \in L_1$ such that $(u)_{i_j} = x_j$ for all $i_j \in N_q' \cap N_p$.*

Proof. The lemma holds due to the facts that $L_1 L_2 = L_2 L_1$ and $p < q$. □

We explain this lemma by the following example.

Example 1 Let $p = 3$ and $q = 7$. Then $N_p = \{1,2,3\}$ and $N_q = \{1,2,\ldots,7\}$. Given $N_q' = \{1,3,4,7\}$ and $x_1, x_2, x_3, x_4 \in X = \Sigma^m$, there is $w \in L_2$ such that $(w)_1 = x_1$, $(w)_3 = x_2$, $(w)_4 = x_3$, and $(w)_7 = x_4$. Then, clearly, there is $u \in L_1$ such that $(u)_1 = x_1$ and $(u)_3 = x_2$, which is illustrated in the diagram below.

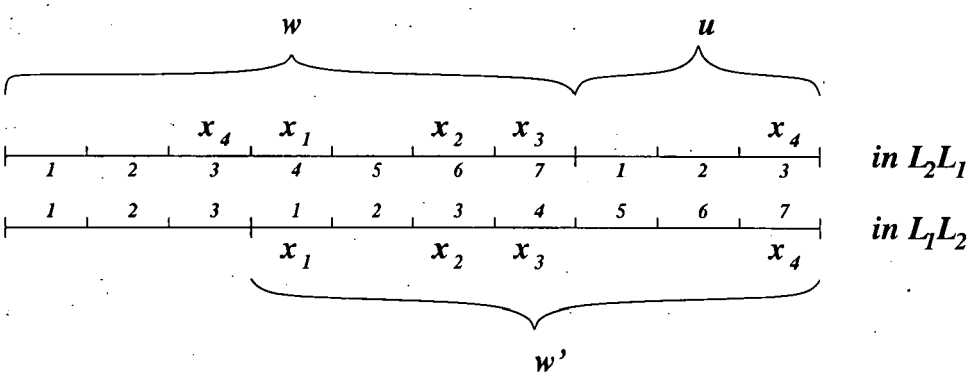


Lemma 7 *Let $N_q' = \{i_1, \ldots, i_n\}$, $1 \le n \le q$, be a subset of N_q and $x_1, \ldots, x_n \in \Sigma^m$. If there exists $w \in L_2$ such that $(w)_{\sigma(i_1)} = x_1, \ldots, (w)_{\sigma(i_n)} = x_n$, then there exists $w' \in L_2$ such that $(w')_{i_1} = x_1, \ldots, (w')_{i_n} = x_n$.*

Proof. Let $w \in L_2$ such that $(w)_{\sigma(i_j)} = x_j$, $i_j \in N'_q$. Then there exists $u \in L_1$ such that $(u)_{\sigma(i_j)} = x_j$ for all $\sigma(i_j) \in N'_q \cap N_p$ by Lemma 6. Since $wu \in L_2L_1$ and $L_2L_1 = L_1L_2$, we have $wu = vw'$ for $v \in L_1$ and $w' \in L_2$. Notice that $(w')_i = (w)_{\sigma(i)}$ for $1 \leq i \leq (q-p)$ and $(w')_i = (u)_{\sigma(i)}$ for $(q-p+1) \leq i \leq q$ due to the fact that $|v| = p$ and the definition of σ. Therefore, $(w')_{i_j} = (w)_{\sigma(i_j)} = x_j$, $1 \leq j \leq n$. □

We explain the above lemma and its proof by the following example.

Example 2 As in the previous example, let $p = 3$ and $q = 7$. Then $N_p = \{1, 2, 3\}$ and $N_q = \{1, 2, \ldots, 7\}$. We are given $N'_q = \{1, 3, 4, 7\}$ and $x_1, x_2, x_3, x_4 \in X = \Sigma^m$, and we know that there is $w \in L_2$ such that $(w)_{\sigma(1)} = x_1$, $(w)_{\sigma(3)} = x_2$, $(w)_{\sigma(4)} = x_3$, and $(w)_{\sigma(7)} = x_4$ (i.e., $(w)_4 = x_1$, $(w)_6 = x_2$, $(w)_7 = x_3$, and $(w)_3 = x_4$). Then, there is $u \in L_1$ such that $(u)_3 = x_4$ by Lemma 6. Since $wu \in L_2L_1 = L_1L_2$, we have $wu = vw'$ for $v \in L_1$ and $w' \in L_2$. We can see from the diagram below that $(w')_1 = x_1$, $(w')_3 = x_2$, $(w')_4 = x_3$, and $(w')_7 = x_4$.



Corollary 1 *Let $N'_q = \{i_1, \ldots, i_n\}$, $1 \leq n \leq q$, be a subset of N_q and $x_1, \ldots, x_n \in \Sigma^m$. If there exists $w \in L_2$ such that $(w)_{\sigma^k(i_1)} = x_1, \ \ldots, \ (w)_{\sigma^k(i_n)} = x_n$, for some k, $0 \leq k \leq q-1$, then there exists $w' \in L_2$ such that $(w')_{i_1} = x_1, \ldots, (w')_{i_n} = x_n$.*

Proof. Apply Lemma 7 k times. □

Theorem 7 *Let $L_1 \subseteq \Sigma^{pm}$, $L_2 \subseteq \Sigma^{qm}$, $p, q, m > 0$, $(p, q) = 1$, and $L_1L_2 = L_2L_1$. Furthermore, let $X = \{w \in \Sigma^m \mid wu \in L_1 \text{ for some } u \in \Sigma^*\}$. Then $L_1 = X^p$ and $L_2 = X^q$.*

Proof. First, we prove that $L_1 \subseteq X^p$ and $L_2 \subseteq X^q$. Define $Y_i = \{w \in \Sigma^m \mid uwv \in L_1 \text{ for } u \in \Sigma^{(i-1)m} \text{ and } v \in \Sigma^{(p-i)m}\}$, $1 \leq i \leq p$,

and $Z_j = \{w \in \Sigma^m \mid uwv \in L_2 \text{ for } u \in \Sigma^{(j-1)m} \text{ and } v \in \Sigma^{(q-j)m}\}$, $1 \leq j \leq q$. Then, clearly, $L_1 \subseteq Y_1 \cdots Y_p$ and $L_2 \subseteq Z_1 \cdots Z_q$. We know that $Y_1 = X$ and it is

obvious that $Z_1 = X$. Let $x \in Z_i$ for some i, $1 \leq i \leq q$. Then there is a word $w \in L_2$ such that $(w)_i = x$. It is clear that $i = \sigma^k(1)$ for some k, $0 \leq k \leq q-1$. Then, by Corollary 1, we know that there is $w' \in L_2$ such that $(w')_1 = x$. So, $x \in Z_1 = X$. Since x is chosen arbitrarily, we have shown that $Z_i \subseteq X$. Similarly, we can show that $Z_i \subseteq X$ for each i, $1 \leq i \leq q$. Therefore, we have $Z_1 \cdots Z_q \subseteq X^q$ and, thus, $L_2 \subseteq X^q$. As a consequence we have that $L_1 \subseteq X^p$.

Second, we prove that $X^p \subseteq L_1$ and $X^q \subseteq L_2$. In order to do so, we prove by induction on n, the cardinality of N'_q, $1 \leq n \leq q$, that for any $N'_q = \{i_1, \ldots, i_n\} \subseteq N_q$ and $x_1, \ldots, x_n \in X$, there exists $w \in L_2$ such that $(w)_{i_k} = x_k$, for $1 \leq k \leq n$. For $n = 1$, let $N'_q = \{i\}$ and x be an arbitrary word in X. If $i = 1$, it is clear that there exists $w \in L_2$ such that $(w)_1 = x$. Otherwise, there $\sigma^k(i) = 1$ for some k, $1 \leq k \leq q-1$. So, by Corollary 1, we have $w \in L_2$ and $(w)_i = x$.
For the induction step, let $N'_q = \{i_1, \ldots, i_n\}$ and $x_1, \ldots, x_n \in X$. Denote $N'_p = N'_q \cap N_p$. If $N'_p \neq \emptyset$ and $N'_q - N'_p \neq \emptyset$, then both $\#N'_p < n$ and $\#(N'_q - N'_p) < n$. Then there exist $u \in L_1$ such that $(u)_{i_k} = x_k$ for $i_k \in N'_p$ and $v \in L_2$ such that $(v)_{\sigma^{q-1}(i_l)} = x_l$ for $i_l \in N'_q - N'_p$ by the induction hypothesis. Since $L_1L_2 = L_2L_1$, we have $uv = wz$ for $w \in L_2$ and $z \in L_1$. Clearly, $(w)_{i_k} = x_k$ for all $i_k \in N'_q$. Otherwise ($N'_p = \emptyset$ or $N'_q - N'_p = \emptyset$), there is an integer $k > 0$ such that $\sigma^k(N'_q)$ satisfies the condition. Then by Corollary 1 and the above arguments, we have $w \in L_2$ such that $(w)_{i_k} = x_k$.

Let $n = q$, i.e., $N'_q = N_q$. Then we have proved that $X^q \subseteq L_2$. Using Lemma 6, we get $X^p \subseteq L_1$. Therefore, we have $L_1 = X^p$ and $L_2 = X^q$. □

5 Further results and open problems

One might be tempted to conjecture that two finite languages F_1 and F_2 commute, $F_1F_2 = F_2F_1$, iff there is a finite language F such that both F_1 and F_2 are unions of powers of F. (Indeed, such a conjecture was presented in [9].) Clearly, if both F_1 and F_2 are of the form

$$F^{i_1} + F^{i_2} + \ldots + F^{i_n},$$

where also $F^0 = \{\lambda\}$ can appear among the terms of the union, then $F_1F_2 = F_2F_1$.

However, the *converse* is not true: F_1 and F_2 may commute without being unions of powers of the same set. The examples used in connection with Theorem 4 can be applied to provide counterexamples. For instance, denote

$$L_1 = a + a^2 + a^3, \quad L_2 = a + a^3, \quad L_3 = \lambda + a.$$

Then

$$L_1L_3 = L_2L_3 (= L_3L_1 = L_3L_2).$$

If we now denote $F_i = L_i + L_3\{b\}L_3$, $i = 1, 2$, it follows that $F_1F_2 = F_2F_1$. It is also clear that F_1 and F_2 cannot be represented as unions of powers of the same

set. (First examples to this effect were given in [3], where it is also shown that the converse holds in case the cardinality of one of the sets F_1 and F_2 is at most 2.)

The validity of the converse, as well as the unique decomposition, can be directly established in some special cases.

For instance, let $\mathcal{E}$ consist of all nonempty finite languages F, where all words of F are of equal length. Then we get immediately the following result.

Lemma 8 *Assume that F is a language in $\mathcal{E}$ and that $F = F_1F_2$, for some F_1 and F_2. Then both F_1 and F_2 are in $\mathcal{E}$.*

Lemma 8 shows that the languages in $\mathcal{E}$ possess a unique prime decomposition and that $\mathcal{E}$ is a free monoid with respect to catenation. Observe that $\mathcal{E}$ is not finitely generated. See [9] for a more detailed discussion.

Thus, the equation $F_1F_2 = F_2F_1$ holds for languages in $\mathcal{E}$ only in case both F_1 and F_2 are powers of the same language X. Moreover, if one of the languages, say F_2, is an arbitrary finite language, we may present it as a (finite) union of languages in $\mathcal{E}$ and use the same argument to show the existence of a language X such that F_1 is a power of X and F_2 is a (finite) union of powers of X. This and other similar results have been established in [10].

The technique in the preceding section was based on more detailed arguments and yields a direct construction of the set X.

In conclusion, we present some general remarks and open problems concerning the *converse* mentioned above. What can be said, in general, about two commuting finite languages F_1 and F_2?

Open problem. Assume that $F_1F_2 = F_2F_1$ holds for two finite languages F_1 and F_2. Characterize the cases, where F_1 and F_2 are *not* unions of powers of the same language. In the sequel we refer to such cases as *exceptional.*

One possible approach to this problem is to consider *positive* decompositions, [9]. As seen above, the ambiguities caused by the presence of λ seem to be the reason behind exceptional cases.

Another approach is is to have an upper bound for the cardinality of one of the two finite languages, say F_1. We already mentioned that if F_1 is of cardinality at most 2 then, independently of F_2, the case is not exceptional, [3]. On the other hand, the example

$$F_1 = a + ab + ba + bb, \qquad F_2 = F_1 + F_1^2 + bab + bbb$$

given in [4] shows that the upper bound 4 for the cardinality of F_1 is not sufficient. It is an open problem whether or not the upper bound 3 is sufficient.

In our few final remarks about commuting, the languages considered are not necessarily finite. Following [4], we say that a finite language $F \subseteq \Sigma^*$ possesses the *Bergman type characterization, BTC* if, for any language $L \subseteq \Sigma^*$ satisfying $FL = LF$, there exists a language $K \subseteq \Sigma^+$ and sets I, J of nonnegative integers such that

$$F = \bigcup_{i \in I} K^i, \qquad L = \bigcup_{j \in J} K^j.$$

(The terminology refers to [1], where the commutation of two polynomials over noncommuting variables is investigated.) It is shown in [4] that every three-word code possesses BTC. We conclude with the following open problems from [4].

Open problem. Does every code possess BTC?

Open problem. Does every three-word language possess BTC?

Acknowledgements We are obliged to the referee for the careful reading of the paper and many valuable suggestions. Discussions with Cezar Campeanu are gratefully acknowledged.

References

[1] Bergman, G.; Centralizers in free associative algebras, *Trans. Amer. Math. Soc.* 137 (1969) 327-344.

[2] Bucher, W., Maurer, H. A., Culik, K., II and Wotschke, D.; Concise description of finite languages, *Theoret. Comput. Sci.* 14 (1981), no. 3, 227–246.

[3] Choffrut, C., Karhumäki, J. and Ollinger, N.; The commutation of finite sets: a challenging problem, TUCS Technical Report 303 (1999), to appear in *Theoret. Comput. Sci.*

[4] Karhumäki, J. and Petre, I.; On the centralizer of a finite set, Springer *LNCS* 1853 (2000) 536-546.

[5] Mateescu, A., Salomaa, A. and Yu, S.; On the decomposition of finite languages, TUCS Technical Report 222 (1998).

[6] Rozenberg, G. and Salomaa, A.; (eds) *Handbook of Formal Languages*, Springer, Berlin, New York, 1997.

[7] Salomaa, A.; *Theory of Automata*, International Series of Monographs in Pure and Applied Mathematics, Vol. 100 Pergamon Press, Oxford-New York-Toronto, 1969.

[8] Salomaa, A.; *Formal Languages*, Academic Press, New York, London, 1973.

[9] Salomaa, A. and Yu, S.; On the decomposition of finite languages, DLT 99 Preproceedings, Aachener Informatik-Berichte 99-5 (1999) 8-20. Appears also in: Rozenberg, G. and Thomas, W.; (eds.) *Developments in Language Theory*, World Scientific, 2000, 22-31.

[10] Shyr, H.J.; *Free Monoids and Languages*, Hon Min Book Company, Taichung, Taiwan R.O.C., 1991.

Received October, 2000

Acta Cybernetica **15** (2002) 353–360.

Reduction of Simple Semi-Conditional Grammars with Respect to the Number of Conditional Productions

Alexander Meduna and Martin Švec*

Abstract

The present paper discusses the descriptional complexity of simple semi-conditional grammars with respect to the number of conditional productions. More specifically, it demonstrates that for every phrase-structure grammar, there exists an equivalent simple semi-conditional grammar that has no more than twelve conditional productions.

Keywords: descriptional complexity, simple semi-conditional grammars

1 Introduction

To describe languages as economically and succintly as possible, formal language theory has recently intensively investigated how to reduce grammars without any decrease of their power (see [1], [4], and [5]). Continuing with this vivid investigation, the present paper discusses the reduction of simple semi-conditional grammars, which characterize the family of recursively enumerable languages (see [2]).

More specifically, besides ordinary context-free productions, simple semi-conditional grammars may have some conditional productions which have an attached string representing a forbidding condition or a permitting condition. This paper concentrates its discussion on the reduction of simple semi-conditional grammars with respect to the number of conditional productions. It demonstrates that for every recursively enumerable language, there exists an equivalent simple semi-conditional grammar that has no more than twelve conditional productions.

2 Definitions

This paper assumes that the reader is familiar with the language theory (see [3]).

Let V be an alphabet. V^* denotes the free monoid generated by V under the operation of concatenation where ε denotes the unit of V^*. Let $V^+ = V^* -$

*Department of Computer Science and Engineering, Brno University of Technology, Božetěchova 2, Brno 61266, Czech Republic

$\{\varepsilon\}$. Given a word, $w \in V^*$, $|w|$ represents the length of w. We set $\text{sub}(w) = \{y : y \text{ is a subword of } w\}$. Given a symbol, $a \in V$, $\#_a w$ denotes the number of occurences of a in w. For $w \in V^+$, first(w) denotes the leftmost symbol of w.

A *semi-conditional grammar* (an *sc*-grammar for short) is a quadruple, $G = (V, T, P, S)$, where V, T and S are the total alphabet, the terminal alphabet ($T \subset V$), and the axiom ($S \in V - T$), respectively, and P is a finite set of productions of the form $(A \to x, \alpha, \beta)$ with $A \in V - T$, $x \in V^*$, $\alpha \in V^+ \cup \{0\}$ and $\beta \in V^+ \cup \{0\}$, where 0 is a special symbol, $0 \notin V$ (intuitively, 0 means that the production's condition is missing). Production $(A \to x, \alpha, \beta) \in P$ is said to be conditional, if $\alpha \neq 0$ or $\beta \neq 0$. G has degree (i, j), where i and j are two natural numbers, if for every $(A \to x, \alpha, \beta) \in P$, $\alpha \in V^+$ implies $|\alpha| \leq i$, and $\beta \in V^+$ implies $|\beta| \leq j$. Let $u, v \in V^*$, and $(A \to x, \alpha, \beta) \in P$. Then, u directly derives v according to $(A \to x, \alpha, \beta)$ in G, denoted by

$$u \Rightarrow_G v \; [(A \to x, \alpha, \beta)]$$

provided for some $u_1, u_2 \in V^*$, the following conditions (a) through (d) hold

(a) $u = u_1 A u_2$,

(b) $v = u_1 x u_2$,

(c) $\alpha \neq 0$ implies $\alpha \in \text{sub}(u)$,

(d) $\beta \neq 0$ implies $\beta \notin \text{sub}(u)$.

When no confusion exists, we simply write $u \Rightarrow_G v$. As usual, we extend $\Rightarrow_G$ to $\Rightarrow_G^i$ (where $i \geq 0$), $\Rightarrow_G^+$, and $\Rightarrow_G^*$. The language of G, denoted by $L(G)$, is defined as $L(G) = \{w \in T^* : S \Rightarrow_G^* w\}$.

Based upon the concept of *sc*-grammars, Meduna and Gopalaratnam [2] have defined a *simple semi-conditional grammar* (an *ssc*-grammar for short) as an *sc*-grammar in which every production has no more than one condition. Formally, let $G = (V, T, P, S)$ be an *sc*-grammar. G is a simple semi-conditional grammar if $(A \to x, \alpha, \beta) \in P$ implies $\{0\} \subseteq \{\alpha, \beta\}$.

3 Results

Theorem 1 *Every recursively enumerable language can be defined by a simple semi-conditional grammar of degree (2,1) with no more than 12 conditional productions.*

Proof. Let L be a recursively enumerable language. By Geffert [1], we can assume that L is generated by a grammar G of the form

$$G = (V, T, P \cup \{AB \to \varepsilon, CD \to \varepsilon\}, S)$$

such that P contains only context-free productions and

$$V - T = \{S, A, B, C, D\}.$$

We construct an *ssc*-grammar G' of degree (2,1) as follows:

$$G' = (V', T, P', S), \quad \text{where}$$
$$V' = V \cup W,$$
$$W = \{\widetilde{A}, \widetilde{B}, \langle\varepsilon_A\rangle, \$, \widetilde{C}, \widetilde{D}, \langle\varepsilon_C\rangle, \#\}, \ V \cap W = \emptyset.$$

The set of productions P' is defined in the following way:

1. if $H \to \alpha \in P$, $H \in V - T$, $\alpha \in V^*$, then add $(H \to \alpha, 0, 0)$ to P';
2. add the following six productions to P':

$$(A \to \widetilde{A}, 0, \widetilde{A}),$$
$$(B \to \widetilde{B}, 0, \widetilde{B}),$$
$$(\widetilde{A} \to \langle\varepsilon_A\rangle, \widetilde{A}\widetilde{B}, 0),$$
$$(\widetilde{B} \to \$, \langle\varepsilon_A\rangle\widetilde{B}, 0),$$
$$(\langle\varepsilon_A\rangle \to \varepsilon, 0, \widetilde{B}),$$
$$(\$ \to \varepsilon, 0, \langle\varepsilon_A\rangle);$$

3. add the following six productions to P':

$$(C \to \widetilde{C}, 0, \widetilde{C}),$$
$$(D \to \widetilde{D}, 0, \widetilde{D}),$$
$$(\widetilde{C} \to \langle\varepsilon_C\rangle, \widetilde{C}\widetilde{D}, 0),$$
$$(\widetilde{D} \to \#, \langle\varepsilon_C\rangle\widetilde{D}, 0),$$
$$(\langle\varepsilon_C\rangle \to \varepsilon, 0, \widetilde{D}),$$
$$(\# \to \varepsilon, 0, \langle\varepsilon_C\rangle).$$

Next, we prove that $L(G') = L(G)$.

Basic idea: Notice that G' has degree (2,1) and contains only 12 conditional productions. The productions of (2) simulate the application of $AB \to \varepsilon$ in G' and the productions of (3) simulate the application of $CD \to \varepsilon$ in G'.

Let us describe the simulation of $AB \to \varepsilon$. First, one occurence of A and one occurence of B are rewritten to $\widetilde{A}$ and $\widetilde{B}$, respectively (no more than one $\widetilde{A}$ and one $\widetilde{B}$ appear in any sentential form). The right neighbor of $\widetilde{A}$ is checked to be $\widetilde{B}$ and $\widetilde{A}$ is rewritten to $\langle\varepsilon_A\rangle$. Then, analogously, the left neighbor of $\widetilde{B}$ is checked to be $\langle\varepsilon_A\rangle$ and $\widetilde{B}$ is rewritten to \$. Finally, $\langle\varepsilon_A\rangle$ and \$ are erased. The simulation of $CD \to \varepsilon$ is analogous.

To establish $L(G) = L(G')$, we first prove the following two claims.

Claim 1 *$S \Rightarrow^*_{G'} x'$ implies $\#_{\widetilde{X}} x' \leq 1$ for all $\widetilde{X} \in \{\widetilde{A}, \widetilde{B}, \widetilde{C}, \widetilde{D}\}$ and some $x' \in (V')^*$.*

Proof. By inspection of productions in P', the only production that can generate $\widetilde{X}$ is of the form $(X \to \widetilde{X}, 0, \widetilde{X})$. This production can be applied only when no $\widetilde{X}$ occurs in the rewritten sentential form. Thus, it is not possible to derive x' from S such that $\#_{\widetilde{X}} x' \geq 2$. □

Informally, next claim says that every occurence of $\langle \varepsilon_A \rangle$ in derivations from S is always followed either by $\widetilde{B}$ or \$, and every occurence of $\langle \varepsilon_C \rangle$ is always followed either by $\widetilde{D}$ or #.

Claim 2 *It holds that*

A) $S \Rightarrow^*_{G'} y'_1\langle \varepsilon_A \rangle y'_2$ *implies* $y'_2 \in (V')^+ \wedge \text{first}(y'_2) \in \{\widetilde{B}, \$\}$ *for any* $y'_1 \in (V')^*$;

B) $S \Rightarrow^*_{G'} y'_1\langle \varepsilon_C \rangle y'_2$ *implies* $y'_2 \in (V')^+ \wedge \text{first}(y'_2) \in \{\widetilde{D}, \#\}$ *for any* $y'_1 \in (V')^*$.

Proof. We establish the proof by the examination of all possible forms of derivations that may occur when deriving a sentential form containing $\langle \varepsilon_A \rangle$ or $\langle \varepsilon_C \rangle$.

A) By the definition of P', the only production that can generate $\langle \varepsilon_A \rangle$ is $p = (\widetilde{A} \to \langle \varepsilon_A \rangle, \widetilde{A}\widetilde{B}, 0)$. This production has the permitting condition $\widetilde{A}\widetilde{B}$, so it can be used provided that $\widetilde{A}\widetilde{B}$ occurs in a sentential form. Furthermore, by Claim 1, no other occurence of $\widetilde{A}$ or $\widetilde{B}$ can appear in the given sentential form. Consequently, we obtain a derivation

$$S \Rightarrow^*_{G'} u'_1\widetilde{A}\widetilde{B}u'_2 \Rightarrow_{G'} u'_1\langle \varepsilon_A \rangle \widetilde{B}u'_2 \ [p]$$

for some $u'_1, u'_2 \in (V')^*$, $\widetilde{A}, \widetilde{B} \notin \text{sub}(u'_1u'_2)$, which represents the only way how to get $\langle \varepsilon_A \rangle$. Obviously, $\langle \varepsilon_A \rangle$ is always followed by $\widetilde{B}$ in $u'_1\langle \varepsilon_A \rangle \widetilde{B}u'_2$.

Next, we discuss how G' can rewrite the subword $\langle \varepsilon_A \rangle \widetilde{B}$ in $u'_1\langle \varepsilon_A \rangle \widetilde{B}u'_2$. There are only two productions having the nonterminals $\langle \varepsilon_A \rangle$ or $\widetilde{B}$ on their left-hand side—$p_1 = (\widetilde{B} \to \$, \langle \varepsilon_A \rangle \widetilde{B}, 0)$ and $p_2 = (\langle \varepsilon_A \rangle \to \varepsilon, 0, \widetilde{B})$. G' cannot use p_2 to erase $\langle \varepsilon_A \rangle$ in $u'_1\langle \varepsilon_A \rangle \widetilde{B}u'_2$ because p_2 forbids an occurence of $\widetilde{B}$ in the string to be rewritten. Production p_1 has also a context condition, but $\langle \varepsilon_A \rangle \widetilde{B} \in \text{sub}(u'_1\langle \varepsilon_A \rangle \widetilde{B}u'_2)$ and thus p_1 can be used to rewrite $\widetilde{B}$ with \$. Hence, we obtain a derivation of the form

$$\begin{array}{lll} S & \Rightarrow^*_{G'} u'_1\widetilde{A}\widetilde{B}u'_2 & \Rightarrow_{G'} u'_1\langle \varepsilon_A \rangle \widetilde{B}u'_2 \quad [p] \\ & \Rightarrow^*_{G'} v'_1\langle \varepsilon_A \rangle \widetilde{B}v'_2 & \Rightarrow_{G'} v'_1\langle \varepsilon_A \rangle \$v'_2 \quad [p_1]. \end{array}$$

Notice that during this derivation, G' may rewrite u'_1 and u'_2 to some v'_1 and v'_2, respectively ($v'_1, v'_2 \in (V')^*$); however, $\langle \varepsilon_A \rangle \widetilde{B}$ remains unchanged after this rewriting.

In this derivation we obtained the second symbol, \$, that can appear as the right neighbor of $\langle \varepsilon_A \rangle$. It suffices to show that there is no other symbol that could appear immediately after $\langle \varepsilon_A \rangle$. By inspection of P', only $(\$ \to \varepsilon, 0, \langle \varepsilon_A \rangle)$ can rewrite \$. However, this production cannot be applied when $\langle \varepsilon_A \rangle$ occurs in the given sentential form. In other words, the occurence of \$ in the subword $\langle \varepsilon_A \rangle \$$

cannot be rewritten before $\langle\varepsilon_A\rangle$ is erased by the production p_2. Hence, $\langle\varepsilon_A\rangle$ is always followed either by $\widetilde{B}$ or \$ and thus the first part of Claim 2 holds.

B) By inspection of productions simulating $AB \to \varepsilon$ and $CD \to \varepsilon$ in G' (see (2) and (3) in the definition of P'), these two sets of productions work analogously. Thus, part B of Claim 2 can be proven by analogy with part A. □

Let us return to the main part of the proof. Let g be a finite substitution from $(V')^*$ to V^* defined as follows:

1. for all $X \in V$: $g(X) = \{X\}$;
2. $g(\widetilde{A}) = \{A\}$, $g(\widetilde{B}) = \{B\}$, $g(\langle\varepsilon_A\rangle) = \{A\}$, $g(\$) = \{B, AB\}$;
3. $g(\widetilde{C}) = \{C\}$, $g(\widetilde{D}) = \{D\}$, $g(\langle\varepsilon_C\rangle) = \{C\}$, $g(\#) = \{C, CD\}$.

Having this substitution, we can now prove the following claim:

Claim 3 *$S \Rightarrow_G^* x$ if and only if $S \Rightarrow_{G'}^* x'$ for some $x \in g(x')$, $x \in V^*$, $x' \in (V')^*$.*

Proof. The claim is proven by induction on the length of derivations.
Only if: We show that

$$S \Rightarrow_G^m x \quad \text{implies} \quad S \Rightarrow_{G'}^* x,$$

where $m \geq 0$, $x \in V^*$; clearly $x \in g(x)$. This is established by induction on m.
Basis: Let $m = 0$. That is, $S \Rightarrow_G^0 S$. Clearly, $S \Rightarrow_{G'}^0 S$.
Induction Hypothesis: Suppose that the claim holds for all derivations of length m or less, for some $m \geq 0$.
Induction Step: Let us consider a derivation $S \Rightarrow_G^{m+1} x$, $x \in V^*$. Since $m + 1 \geq 1$, there is some $y \in V^+$ and $p \in P \cup \{AB \to \varepsilon, CD \to \varepsilon\}$ such that $S \Rightarrow_G^m y \Rightarrow_G x$ $[p]$. By the induction hypothesis, there is a derivation $S \Rightarrow_{G'}^* y$.

There are three cases that cover all possible forms of the production p:

(i) $p = H \to y_2 \in P$, $H \in V - T$, $y_2 \in V^*$. Then, $y = y_1 H y_3$ and $x = y_1 y_2 y_3$, $y_1, y_3 \in V^*$. Because we have $(H \to y_2, 0, 0) \in P'$, $S \Rightarrow_{G'}^* y_1 H y_3 \Rightarrow_{G'} y_1 y_2 y_3$ $[(H \to y_2, 0, 0)]$ and $y_1 y_2 y_3 = x$.

(ii) $p = AB \to \varepsilon$. Then, $y = y_1 AB y_3$ and $x = y_1 y_3$, $y_1, y_3 \in V^*$. In this case, there is the following derivation:

$$\begin{array}{lll}
S & \Rightarrow_{G'}^* & y_1 A B y_3 \\
& \Rightarrow_{G'} & y_1 \widetilde{A} B y_3 \quad [(A \to \widetilde{A}, 0, \widetilde{A})] \\
& \Rightarrow_{G'} & y_1 \widetilde{A} \widetilde{B} y_3 \quad [(B \to \widetilde{B}, 0, \widetilde{B})] \\
& \Rightarrow_{G'} & y_1 \langle\varepsilon_A\rangle \widetilde{B} y_3 \quad [(\widetilde{A} \to \langle\varepsilon_A\rangle, \widetilde{A}\widetilde{B}, 0)] \\
& \Rightarrow_{G'} & y_1 \langle\varepsilon_A\rangle \$ y_3 \quad [(\widetilde{B} \to \$, \langle\varepsilon_A\rangle \widetilde{B}, 0)] \\
& \Rightarrow_{G'} & y_1 \$ y_3 \quad [(\langle\varepsilon_A\rangle \to \varepsilon, 0, \widetilde{B})] \\
& \Rightarrow_{G'} & y_1 y_3 \quad [(\$ \to \varepsilon, 0, \langle\varepsilon_A\rangle)].
\end{array}$$

(iii) $p = CD \to \varepsilon$. Then, $y = y_1CDy_3$ and $x = y_1y_3$, $y_1, y_3 \in V^*$. By analogy with (ii), there exists the derivation $S \Rightarrow^*_{G'} y_1CDy_3 \Rightarrow^6_{G'} y_1y_3$.

If: By induction on the length n of derivations in G', we prove that

$$S \Rightarrow^n_{G'} x' \quad \text{implies} \quad S \Rightarrow^*_G x$$

for some $x \in g(x')$, $x \in V^*$, $x' \in (V')^*$.
Basis: Let $n = 0$. That is, $S \Rightarrow^0_{G'} S$. It is obvious that $S \Rightarrow^0_G S$ and $S \in g(S)$.
Induction Hypothesis: Assume that the claim holds for all derivations of length n or less, for some $n \geq 0$.
Induction Step: Consider a derivation $S \Rightarrow^{n+1}_{G'} x'$, $x' \in (V')^*$. Since $n + 1 \geq 1$, there is some $y' \in (V')^+$ and $p' \in P'$ such that $S \Rightarrow^n_{G'} y' \Rightarrow_{G'} x'$ $[p']$, and by the induction hypothesis, there is also a derivation $S \Rightarrow^*_G y$ such that $y \in g(y')$.

By inspection of P', the following cases (i) through (xiii) cover all possible forms of p':

(i) $p' = (H \to y_2, 0, 0) \in P'$, $H \in V - T$, $y_2 \in V^*$. Then, $y' = y'_1Hy'_3$, $x' = y'_1y_2y'_3$, $y'_1, y'_3 \in (V')^*$ and y has the form $y = y_1Zy_3$, where $y_1 \in g(y'_1)$, $y_3 \in g(y'_3)$ and $Z \in g(H)$. Because for all $X \in V - T : g(X) = \{X\}$, the only Z is H and thus $y = y_1Hy_3$. By the definition of P' (see (1)), there exists a production $p = H \to y_2$ in P and we can construct the derivation $S \Rightarrow^*_G y_1Hy_3 \Rightarrow_G y_1y_2y_3$ $[p]$ such that $y_1y_2y_3 = x$, $x \in g(x')$.

(ii) $p' = (A \to \widetilde{A}, 0, \widetilde{A})$. Then, $y' = y'_1Ay'_3$, $x' = y'_1\widetilde{A}y'_3$, $y'_1, y'_3 \in (V')^*$ and $y = y_1Zy_3$, where $y_1 \in g(y'_1)$, $y_3 \in g(y'_3)$ and $Z \in g(A)$. Because $g(A) = \{A\}$, the only Z is A, so we can express $y = y_1Ay_3$. Having the derivation $S \Rightarrow^*_G y$ such that $y \in g(y')$, it is easy to see that also $y \in g(x')$ because $A \in g(\widetilde{A})$.

(iii) $p' = (B \to \widetilde{B}, 0, \widetilde{B})$. By analogy with (ii), $y' = y'_1By'_3$, $x' = y'_1\widetilde{B}y'_3$, $y = y_1By_3$, where $y'_1, y'_3 \in (V')^*$, $y_1 \in g(y'_1)$, $y_3 \in g(y'_3)$ and thus $y \in g(x')$ because $B \in g(\widetilde{B})$.

(iv) $p' = (\widetilde{A} \to \langle\varepsilon_A\rangle, \widetilde{A}\widetilde{B}, 0)$. By the permitting condition of this production, $\widetilde{A}\widetilde{B}$ surely occurs in y'. By Claim 1, no more than one $\widetilde{A}$ can occur in y'. Therefore, y' must be of the form $y' = y'_1\widetilde{A}\widetilde{B}y'_3$, where $y'_1, y'_3 \in (V')^*$ and $\widetilde{A} \notin \text{sub}(y'_1y'_3)$. Then, $x' = y'_1\langle\varepsilon_A\rangle\widetilde{B}y'_3$ and y is of the form $y = y_1Zy_3$, where $y_1 \in g(y'_1)$, $y_3 \in g(y'_3)$ and $Z \in g(\widetilde{A}\widetilde{B})$. Because $g(\widetilde{A}\widetilde{B}) = \{AB\}$, the only Z is AB; thus, we obtain $y = y_1ABy_3$. By the induction hypothesis, we have a derivation $S \Rightarrow^*_G y$ such that $y \in g(y')$. According to the definition of g, $y \in g(x')$ as well because $A \in g(\langle\varepsilon_A\rangle)$ and $B \in g(\widetilde{B})$.

(v) $p' = (\widetilde{B} \to \$, \langle\varepsilon_A\rangle\widetilde{B}, 0)$. This production can be applied provided that $\langle\varepsilon_A\rangle\widetilde{B} \in \text{sub}(y')$. Moreover, by Claim 1, $\#_{\widetilde{B}}y' \leq 1$. Hence, we can express $y' = y'_1\langle\varepsilon_A\rangle\widetilde{B}y'_3$, where $y'_1, y'_3 \in (V')^*$ and $\widetilde{B} \notin \text{sub}(y'_1y'_3)$. Then, $x' = y'_1\langle\varepsilon_A\rangle\$y'_3$ and $y = y_1Zy_3$, where $y_1 \in g(y'_1)$, $y_3 \in g(y'_3)$ and

$Z \in g(\langle\varepsilon_A\rangle\widetilde{B})$. By the definition of g, $g(\langle\varepsilon_A\rangle\widetilde{B}) = \{AB\}$, so $Z = AB$ and $y = y_1ABy_3$. By the induction hypothesis, we have a derivation $S \Rightarrow_G^* y$ such that $y \in g(y')$. Because $A \in g(\langle\varepsilon_A\rangle)$ and $B \in g(\$)$, $y \in g(x')$ as well.

(vi) $p' = (\langle\varepsilon_A\rangle \to \varepsilon, 0, \widetilde{B})$. Application of $(\langle\varepsilon_A\rangle \to \varepsilon, 0, \widetilde{B})$ implies that $\langle\varepsilon_A\rangle$ occurs in y'. Claim 2 says that $\langle\varepsilon_A\rangle$ has either $\widetilde{B}$ or \$ as its right neighbor. Since the forbidding condition of p' forbids an occurence of $\widetilde{B}$ in y', the right neighbor of $\langle\varepsilon_A\rangle$ must be \$. As a result, we obtain $y' = y_1'\langle\varepsilon_A\rangle\$y_3'$ where $y_1', y_3' \in (V')^*$. Then, $x' = y_1'\$y_3'$ and y is of the form $y = y_1Zy_3$, where $y_1 \in g(y_1')$, $y_3 \in g(y_3')$ and $Z \in g(\langle\varepsilon_A\rangle\$)$. By the definition of g, $g(\langle\varepsilon_A\rangle\$) = \{AB, AAB\}$. If $Z = AB$, $y = y_1ABy_3$. Having the derivation $S \Rightarrow_G^* y$, it holds that $y \in g(x')$ because $AB \in g(\$)$.

(vii) $p' = (\$ \to \varepsilon, 0, \langle\varepsilon_A\rangle)$. Then, $y' = y_1'\$y_3'$ and $x' = y_1'y_3'$, where $y_1', y_3' \in (V')^*$. Express $y = y_1Zy_3$ so that $y_1 \in g(y_1')$, $y_3 \in g(y_3')$ and $Z \in g(\$)$, where $g(\$) = \{B, AB\}$. Let $Z = AB$. Then, $y = y_1ABy_3$ and there exists the derivation $S \Rightarrow_G^* y_1ABy_3 \Rightarrow_G y_1y_3\ [AB \to \varepsilon]$, where $y_1y_3 = x$, $x \in g(x')$.

In cases (ii) through (vii) we discussed all six productions simulating the application of $AB \to \varepsilon$ in G' (see (2) in the definition of P'). Cases (viii) – (xiii) should cover productions simulating the application of $CD \to \varepsilon$ in G' (see (3)). However, by inspection of these two sets of productions, it is easy to see that they work analogously. Therefore, we leave this part of the proof to the reader (it can be established by analogy with (ii) – (vii) by replacing nonterminals $A, B, \widetilde{A}, \widetilde{B}, \langle\varepsilon_A\rangle$ and \$ with $C, D, \widetilde{C}, \widetilde{D}, \langle\varepsilon_C\rangle$ and #).

We have completed the proof and established Claim 3 by the principle of induction. □

Observe that $L(G) = L(G')$ follows from Claim 3. Indeed, according to the definition of g, we have $g(a) = \{a\}$ for all $a \in T$. Thus, from Claim 3, we have for any $x \in T^*$:

$$S \Rightarrow_G^* x \quad \text{if and only if} \quad S \Rightarrow_{G'}^* x.$$

Consequently, $L(G) = L(G')$ and the theorem holds. ■

In fact, the previous proof established more than stated in Theorem 1. Indeed, it also reduced the number of nonterminals as the next corollary says.

Corollary 1 *Every recursively enumerable language can be generated by a simple semi-conditional grammar of degree (2,1) with no more than 12 conditional productions and 13 nonterminals.*

Proof. Observe that G' has 13 nonterminals in the proof of Theorem 1. □

The above corollary tells us that besides the number of conditional productions, we have also reduced the semi-conditional grammars with respect to the number of nonterminals. In addition, we were able to establish this result for semi-conditional grammars of degree (2,1). This result gives rise to a question of whether we can

further reduce the number of conditional productions in the semi-conditional grammars of any degree. In other words, consider the semi-conditional grammars with productions having context conditions of any length. Can they generate any recursively enumerable language with fewer than 12 conditional productions?

Acknowledgment

The authors thank the anonymous referee for several useful comments regarding the first version of this paper.

References

[1] Geffert, V.: How to generate languages using only two pairs of parentheses. J. Inform. Process. Cybernet, EIK27, 781–786 (1990).

[2] Meduna, A., and Gopalaratnam, A.: On Semi-Conditional Grammars with Productions Having either Forbidding or Permitting Conditions, Acta Cybernetica 11 (1994), 307–323.

[3] Meduna, A.: Automata and Languages: Theory and Applications. Springer, London, 2000.

[4] Proceedings of the International Workshop on Descriptional Complexity of Automata, Grammars and Related Structures, July 20–23, 1999, Ed. by J. Dassow and D. Wotschke, Otto-von-Guericke University, Magdeburg, Germany, 1999.

[5] Proceedings of the Second Workshop on Descriptional Complexity of Automata, Grammars and Related Structures, July 27–29, 2000, London, Canada.

Received February, 2001

Acta Cybernetica **15** (2002) 361–367.

Closed On-Line Bin Packing

E. Asgeirsson[*], U. Ayesta[†], E. Coffman[‡], J. Etra[§], P. Momčilović[‡], D. Phillips[*], V. Vokhshoori[‡], Z. Wang[¶] and J. Wolfe[‡]

Abstract

An optimal algorithm for the classical bin packing problem partitions (packs) a given set of items with sizes at most 1 into a smallest number of unit-capacity bins such that the sum of the sizes of the items in each bin is at most 1. Approximation algorithms for this NP-hard problem are called *on-line* if the items are packed sequentially into bins with the bin receiving a given item being independent of the number and sizes of all items as yet unpacked. *Off-line* algorithms plan packings assuming full (advance) knowledge of all item sizes. The *closed* on-line algorithms are intermediate: item sizes are not known in advance but the number n of items is. The uniform model, where the n item sizes are independent uniform random draws from [0,1], commands special attention in the average-case analysis of bin packing algorithms. In this model, the expected wasted space produced by an optimal off-line algorithm is $\Theta(\sqrt{n})$, while that produced by an optimal on-line algorithm is $\Theta(\sqrt{n \log n})$. Surprisingly, an optimal closed on-line algorithm also wastes only $\Theta(\sqrt{n})$ space on the average. A proof of this last result is the principal contribution of this paper. However, we also identify a class of optimal closed algorithms, extend the main result to other probability models, and give an estimate of the hidden constant factor.

1 Introduction

An instance of the one-dimensional bin packing problem is a list $L_n = \langle a_1, a_2, \ldots, a_n \rangle$ of *items* that must be packed into, i.e., partitioned among, a minimum-cardinality set of *bins* $B_1, B_2, \ldots$ subject to the constraint that the set of items in any bin fits within that bin's capacity. In the usual way, we will take the bin capacity to be 1 for convenience, so a set of items fits into a bin if and only if the item sizes sum to no more than 1. The unused space in B_i is called a *gap*

[*]Department of Industrial Engineering and Operations Research, Columbia University, New York, NY 10027

[†]INRIA, 2004 Route des Lucioles, Sophia Antipolis Cedex, France

[‡]Department of Electrical Engineering, Columbia University, New York, NY 10027

[§]Columbia Law School, Columbia University, New York, NY 10027

[¶]Graduate School of Business, Columbia University, New York, NY 10027

and is denoted by g_i. The sum of the gaps in the occupied bins of a packing is the *wasted space* of the packing.

The bin packing problem has countless applications in operations research and engineering. To name just a few, we mention storage allocation for computer networks, assigning advertisements to newspaper columns, assigning commercials to station breaks on television, writing a collection of files to several floppy disks, packing trucks with a given weight limit, and the cutting-stock problems of various industries like those producing lumber and cable.

Let A denote an arbitrary approximation algorithm for the NP-hard bin packing problem and let OPT denote an algorithm that produces optimal packings. Let $A(L_n)$ and $\mathrm{OPT}(L_n)$ denote the numbers of bins used by algorithms A and OPT. In the classical analysis of bin packing approximation algorithms, combinatorial methods are used to derive worst-case performance ratios

$$R_A := \sup_L \{A(L)/\mathrm{OPT}(L)\}$$

and their asymptotic variants. Less often, probabilistic studies that are typically quite difficult are conducted in order to obtain average-case performance. The average-case approach is followed in this paper. *The item-size distribution is taken to be the uniform distribution on* $[0,1]$, denoted as usual by $U(0,1)$. This is the distribution of choice in bin packing analysis, along with the assumption that item sizes are independent. For general coverage of the probabilistic analysis of bin packing algorithms, see the monograph by Coffman and Lueker(1991).

A bin packing algorithm is called *on-line* if it packs every item a_i solely on the basis of the sizes of the items a_j, $1 \le j \le i$, i.e., without any information on subsequent items. The decisions of an on-line algorithm are irrevocable; packed items cannot be repacked at later times. Two classical on-line algorithms are First Fit and Best Fit. Each of these algorithms begins by putting a_1 into B_1. Thereafter, First Fit places the next item into the lowest indexed (first) gap no smaller than the item, and Best Fit puts the next item into a smallest gap no smaller than the item with ties resolved in favor of the lowest indexed bins.

A bin packing algorithm that can use full knowledge of all items in packing L_n is called *off-line*. One of the first results in the average-case theory was a proof by Lueker(1982) that an optimal off-line algorithm has the following asymptotic bound.

$$E\mathrm{OPT}_{offline}(L_n) = \frac{n}{2} + \Theta(\sqrt{n}).$$

where $\Theta(\sqrt{n})$ bounds the expected wasted space, since the expected total item size gives the $n/2$ term. More recently, Shor(1991) proved that on-line packings must produce greater expected wasted space by at least a log factor. In particular, he showed that

$$E\mathrm{OPT}_{online}(L_n) = \frac{n}{2} + \Theta(\sqrt{n \log n}).$$

Although there is no known simple algorithm for achieving this bound, the Best

Fit (BF) algorithm comes close in that (see Shor(1986))

$$E\mathrm{BF}(L_n) = \frac{n}{2} + \Theta(\sqrt{n}\log^{3/4} n)$$

The *closed* on-line algorithms are intermediate between the classes of on-line and off-line algorithms: item sizes are not known in advance, but the number n of items is. As noted by Shor(1986), it is surprising that one can produce an algorithm that achieves $O(\sqrt{n})$ expected wasted space without knowing item sizes in advance. However, the algorithm must know n, i.e., it must be a closed on-line algorithm. According to one such algorithm, which we call Closed Best Fit (CBF), the first $\lceil n/2 \rceil$ items are packed one to a bin and the remaining $\lfloor n/2 \rfloor$ items are packed by Best Fit. The claim is that CBF wastes at most $O(\sqrt{n})$ space on average, and so the following bound on closed on-line packing holds.

Theorem 1

$$E\mathrm{OPT}_{closed}(L_n) = \frac{n}{2} + \Theta(\sqrt{n})$$

We have seen no proof of this result, and while it is true that standard techniques may be applied in such a proof, the way in which they are applied has novel features. For this reason, and because the improvement possible in closed on-line bin packing is indeed unexpected, the next section sets down for the record a proof of Theorem 1. Still more reasons are provided by the additional results to which the analysis leads. For example, we derive a compact upper bound on the hidden constant factor from the analysis of a random walk. Further, as discussed in Section 3, Theorem 1 will be seen to apply to a number of practical matching algorithms, and to be extendible to distributions other than the uniform.

2 Proof of Theorem 1

For convenience, we assume hereafter that n is even; this will not affect our asymptotic results. Let $L_n^{(1)}$ and $L_n^{(2)}$ be the sublists of the first $n/2$ and last $n/2$ items of L_n, respectively. We begin by proving $O(\sqrt{n})$ wasted space for the modification of CBF which closes any bin B_j, $j \leq n/2$, after it receives a second item, and closes any bin B_j, $j > n/2$, after it receives its first item. Denote the modified algorithm by CBF_*. An example is shown in Figure 1(a). After proving that Theorem 1 holds for CBF_*, we will show that $\mathrm{CBF}_*(L_n) \geq \mathrm{CBF}(L_n)$ for all L_n, thus completing the proof of Theorem 1.

We begin with a key property of CBF_* packings.

Lemma 1 *Let L_n and L'_n differ only in the permutations of their last $n/2$ items. Then $CBF_*(L_n) = CBF_*(L'_n)$.*

Proof. Consider the *ordered* CBF_* packing of L_n in which the bins are arranged so that the first $n/2$ items are in decreasing size order, as illustrated in Figure 1(b). We say that this packing is a *canonical* packing if in addition the last $n/2$ items

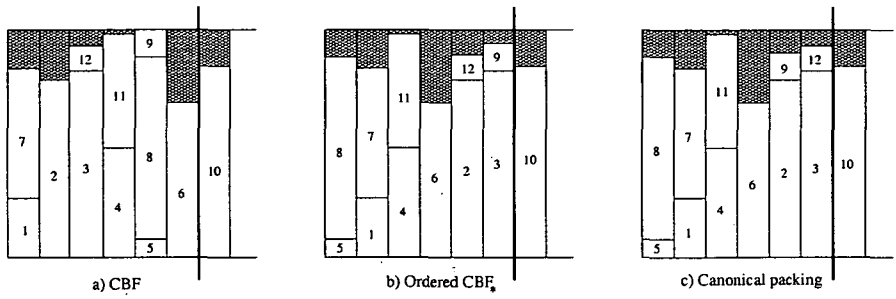


Figure 1: An example with items 1 through $n = 12$ having sizes

$$.26, .78, .82, .48, .08, .68, .57, .8, .12, .84, .5, .11.$$

are in *increasing* size order. Roughly speaking, CBF_* attempts to pack bins with matched items, one large and one small. However, ordered matchings (packings) are not necessarily canonical unless $L_n^{(2)}$ is in increasing size order. For example, in Figure 1(b) a canonical matching requires that items a_9 and a_{12} be interchanged.

On the other hand, the CBF_* packing can be put into canonical form without changing $\mathrm{CBF}_*(L_n)$. To see this, suppose items i_1, i_2 are in $L_n^{(1)}$ and matched with j_1, j_2, respectively, in bins of the ordered CBF_* packing. If $a_{j_1} > a_{j_2}$ and $a_{i_1} > a_{i_2}$ then $a_{j_1} + a_{i_2} \leq a_{j_1} + a_{i_1} \leq 1$ and $a_{j_2} + a_{i_1} \leq a_{j_1} + a_{i_1} \leq 1$, so we can interchange items j_1 and j_2 without exceeding bin capacity. Iterating these interchanges at most $O(n^2)$ times brings the CBF_* packing into canonical form up through $B_{n/2}$. Trivially, the items in the singleton bins beyond $B_{n/2}$ can then be sorted into increasing order, at which point the entire packing is a canonical packing. In addition, the set of items in singleton bins can not have changed, since the Best Fit rule depends only on gap sizes and not on the bin (gap) indexing. We conclude that the cardinality of the CFB_* packing is left unchanged at the end of the ordering process. It remains only to observe that CBF_* packings for lists L_n and L'_n that differ only in the permutation of their last $n/2$ items will converge under the ordering process to the same equal-cardinality canonical packing. □

We now prove that Theorem 1 holds for CBF_*, and in the process, find the hidden constant factor.

Lemma 2

$$E\mathrm{CBF}_*(L_n) - \frac{n}{2} \sim \sqrt{\frac{\pi n}{8}}$$

as $n \to \infty$.

Proof. By Lemma 1 we need consider only canonical CBF_* packings. Let $N_1(y)$ be the number of items in $L_n^{(1)}$ with sizes less than y and let $N_2(y)$ be the number of

items in $L_n^{(2)}$ with sizes greater than $1-y$. Define

$$\delta(y) := N_2(y) - N_1(y)$$

and note that $\max_{0\le y\le 1} \delta(y) \ge \delta(0) = 0$. It is easy to see that

$$\mathrm{CBF}_*(L_n) = \frac{n}{2} + \max_{0\le y\le 1} \delta(y). \tag{1}$$

As an example, note that $\max_{0\le y\le 1} \delta(y) = 1$ in Figure 1(b), and that $\delta(y)$ achieves its maximum for any $y \in (1-a_{11}, a_6)$. To verify (1), one can argue in terms of the number of singleton bins beyond $B_{n/2}$ in the CBF_* packing, which is just $\mathrm{CBF}_*(L_n) - n/2$. To the right of the rightmost singleton bin B_j with $j \le n/2$, the number of items from $L_n^{(2)}$ less the number of items from $L_n^{(1)}$ gives the maximum of $\delta(y)$ over $[0,1]$ and is equal to the number of singleton bins beyond $B_{n/2}$.

We now interpret $\delta(y)$ as a random walk that evolves as y increases from 0 to 1. For each size a in $L_n^{(2)}$ a plus is plotted at point a, and for each size a in $L_n^{(1)}$ a minus is plotted at point $1-a$. For each minus encountered as y increases from 0 to 1, $\delta(y)$ steps down by 1, and for each plus encountered, $\delta(y)$ steps up by 1. Let δ_i, $0 \le i \le n$, be the position of this random walk after the ith jump. As can be seen, $\{\delta_i\}$ is a classical n-step symmetric random walk with the constraint that its paths start and end at the origin, i.e., $\delta_0 = \delta_n = 0$. Letting $n = 2\nu$, the number of such paths is $\binom{2\nu}{\nu}$. By the reflection principle (see e.g, Feller(1968), p. 72), the number of such paths that hit or exceed k is $\binom{2\nu}{\nu+k}$, $0 \le k \le \nu$, and so

$$E \max_{0\le i\le 2\nu} \delta_i = \frac{1}{\binom{2\nu}{\nu}} \sum_{1\le k\le \nu} \binom{2\nu}{\nu+k}.$$

By the binomial theorem, the sum evaluates to $\frac{1}{2}(2^{2\nu} - \binom{2\nu}{\nu})$, so routine applications of Stirling's formula yield

$$E \max_{0\le y\le 1} \delta(y) = E \max_{0\le i\le n} \delta_i \sim \sqrt{\frac{\pi n}{8}}$$

as $n \to \infty$. □

We will be done once we have proved

Lemma 3

$$\mathrm{CBF}_*(L_n) \ge \mathrm{CBF}(L_n).$$

Proof. Let $a_{\ell_1}, \ldots, a_{\ell_k}$ be the subsequence of items in $L_n^{(2)}$ that are packed by CBF into bins B_j, $j \le n/2$, that already have at least two items, or bins B_j, $j > n/2$, that already have at least one item. Remove these items from the CBF packing and repack them best-fit into the singleton bins B_j, $j \le n/2$. That is, item a_{ℓ_i} is put into a singleton bin B_j, $j \le n/2$, with the smallest gap no smaller than $1 - a_{\ell_i}$, if such a bin exists; if no such bin exists, a_{ℓ_i} is put into an empty bin (necessarily

beyond $B_{n/2}$). In either case, the bin receiving a_{ℓ_i} is then closed. The final packing is a CBF_* packing of a list L'_n that can differ from L_n only in the permutation of the last $n/2$ items. Moreover, the final packing has a cardinality at least that of the original CBF packing. In particular, $\mathrm{CBF}_*(L'_n) - \mathrm{CBF}(L_n) \geq 0$ is the number of new singleton bins produced in the new packing. By Lemma 1 we can then conclude that

$$\mathrm{CBF}_*(L'_n) = \mathrm{CBF}_*(L_n) \geq \mathrm{CBF}(L_n)$$

which proves the lemma. □

3 Final Remarks

Consider the closed on-line algorithm that (i) packs the first $n/2$ items one to a bin, (ii) sorts the bins so that the items are in decreasing order, and (iii) packs the remaining items First Fit. This is the algorithm actually proposed by Shor(1986). Let us call this algorithm Closed First Fit (CFF) and define CFF_* just as we defined the variant CBF_* of CBF (limiting bins to at most two items). In comparing CFF_* and CBF_*, we observe that packing best fit is like packing first fit into a decreasing sequence, so the two algorithms give, for all L_n, exactly the same packing.

Theorem 1 is easily generalized to any distribution symmetric around $1/2$ that is not concentrated entirely at $1/2$. Further, we can apply the same ideas to distributions $U(0, 1/p)$, with p an integer. For example, suppose $p = 3$. Then we take $n/6$ bins and divide each into thirds. The top thirds of these bins are packed as before as if they were bins themselves; only the scaling by a factor of 3 has any effect. Similarly, the middle thirds are packed after top thirds and then the bottom thirds are packed last. Bins beyond $B_{n/6}$ are introduced as needed and packed as if they consisted of 3 bins with capacity $1/3$. The extension of Theorem 1 follows easily.

References

[1] Coffman, E. G., Garey, M. R., and Johnson, D. S.: (1995), Approximation algorithms for bin packing: A survey, in *Approximation algorithms for NP-hard problems*, D. Hochbaum (ed.), PWS Publishing Co., Boston.

[2] Coffman, E. G. and Lueker, G. S.: (1991), *Probabilistic Analysis of Packing and Partitioning Algorithms*, Wiley, New York.

[3] Csirik, J., Frenk, J. B. G., Galambos, G., and Rinnooy Kan, A. H. G.: (1991), Probabilistic analysis of algorithms for dual bin packing problems, *J. Algorithms*, **12**, 189–203.

[4] Feller, W.: (1968), *Probability Theory and Its Applications, Vol. 1*, Wiley, New York.

[5] Lueker, G. S.: (1982), An average-case analysis of bin packing with uniformly distributed item sizes, *Technical Report 181*, University of California at Irvine, Department of Information and Computer Science.

[6] Shor, P. W.: (1986), The average-case analysis of some on-line algorithms for bin packing, *Combinatorica*, **6**, 179–200.

[7] Shor, P. W.: (1991), How to pack better than Best Fit: Tight bounds for average-case on-line bin packing, *Proc. 32nd Ann. Symp. Found. Comp. Sci.*, 752–759, IEEE Comp. Soc. Press, New York.

Received January, 2002

Acta Cybernetica **15** (2001) 369–378.

A PTAS for single machine scheduling with controllable processing times

Petra Schuurman* and Gerhard J. Woeginger†

Abstract

We deal with a single machine scheduling problem in which each job has a release date, a delivery time and a controllable processing time. The fact that the jobs have a controllable processing time means that it is allowed to compress (a part of) the processing time of the job, in return for compression cost. The objective is to find a schedule that minimizes the total cost, that is, the latest delivery time of any job plus the total compression cost. In this note we discuss how the techniques of Hall and Shmoys [3] and Hall [1] can directly be applied to design a polynomial time approximation scheme for this problem.

Keywords. Scheduling, worst case analysis, approximation algorithm, approximation scheme, controllable processing time.

1 Introduction

We consider a scheduling problem in which n jobs, $J_1, \ldots, J_n$, have to be scheduled on a single machine. Each job J_j has a processing requirement p_j and it becomes available for processing at a specific point in time, which we call its release date r_j. After its processing, J_j needs some delivery time (independent of the machine) before it is completed (e.g. cooling off or transportation time); we denote this delivery time by q_j. We assume that no *preemption* is allowed, i.e. once a job has been started, it must be completed without interruptions. The goal is to minimize the latest job delivery completion time, which we call the *length* of the schedule. This scheduling problem with release dates and delivery times is usually denoted by $1 \mid r_j \mid L_{\max}$.

In this paper we consider a more difficult variation of problem $1 \mid r_j \mid L_{\max}$: There are situations where one can and wishes to board out part of the work. In case part of the work of job J_j is boarded out, we say that J_j is *compressed.* The

*Email: petra@win.tue.nl. Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands.

†Email: gwoegi@opt.math.tu-graz.ac.at. Institut für Mathematik B, TU Graz, Steyrergasse 30, A-8010 Graz, Austria, and Department of Mathematics, University of Twente, 7500 AE Enschede, The Netherlands. Supported by the START program Y43-MAT of the Austrian Ministry of Science.

amount J_j is compressed is denoted by x_j, and the maximum possible compression of J_j is denoted by u_j. Here x_j is a decision variable with $0 \leq x_j \leq u_j$; note that x_j does not need to be integral. We call the amount of processing time of a job J_j after compression, its *shortened processing time* a_j. Clearly, $a_j = p_j - x_j$. Of course compressing a job J_j results in extra costs, so-called *compression costs*. We denote the compression cost per unit of J_j by c_j. The total compression cost, denoted by C, satisfies $C = \sum_{j=1}^{n} c_j x_j$. The objective is now to find a schedule σ that minimizes the total cost $T(\sigma)$, that is the length $L(\sigma)$ of the schedule plus the total compression cost $C(\sigma)$.

For the computational complexity of the single-machine problem, the following is known. Let us first discuss the cases where all delivery times are equal. Then determining a schedule with minimal length of an instance of $1 \,|\, r_j \,|\, L_{\max}$ comes down to finding a schedule with minimal makespan, where the makespan is the completion time of the latest job. In this case, determining a schedule with minimal makespan can be done in polynomial time by ordering the jobs in nondecreasing order of release date. The compressible processing time variant also admits a polynomial time algorithm in case of equal delivery times: After sorting the jobs in nondecreasing order of release date, we start compressing the jobs with compression cost less than 1 in order of nonincreasing compression costs, where we only compress a job in case the length of the schedule thereby decreases. Now let us turn to the cases with arbitrary delivery times. Lenstra, Rinnooy Kan & Brucker [5] proved that $1 \,|\, r_j \,|\, L_{\max}$ is $\mathcal{NP}$-hard in the strong sense. Therefore, the single machine problem with compressible processing times, which encloses $1 \,|\, r_j \,|\, L_{\max}$ as a special case, is also strongly $\mathcal{NP}$-hard.

These later observations justify the search for approximate solutions by means of polynomial time approximation algorithms. We say that such an approximation algorithm has *worst case performance guarantee* ρ, or is a ρ-approximation algorithm for short, if it always delivers a solution with value at most $\rho \cdot \mathrm{OPT}$. Here, OPT denotes the value of an optimal solution, which will also be denoted by T^* in our case. For a strongly $\mathcal{NP}$-hard problem, the best that we can hope for, is the existence of a polynomial time approximation scheme, a PTAS for short. A PTAS is a family of polynomial time $(1+\varepsilon)$-approximation algorithms for all $\varepsilon > 0$.

Already in 1971 Schrage [8] developed a 2-approximation algorithm for $1 \,|\, r_j \,|\, L_{\max}$ based on a simple heuristic. During the eighties various improvements upon Schrage's heuristic were designed to obtain better performance guarantees. First Potts [7] modified the heuristic of Schrage into a $\frac{3}{2}$-approximation algorithm, and then Hall and Shmoys [3] on their turn improved Potts' heuristic to get a $\frac{5}{4}$-approximation algorithm. In 1989, Hall and Shmoys [2] developed a new idea, that, among others, helped them in designing an approximation scheme for $1 \,|\, r_j \,|\, L_{\max}$. Hereby the approximability status of $1 \,|\, r_j \,|\, L_{\max}$ was determined. The results of Hall and Shmoys [2] were even stronger, since they extended to the problem with precedence constraints.

In 1991, Zdrzalka [9] designed a $(\frac{1}{2}+\tau)$-approximation algorithm for the single-machine problem with compressible processing times. Here, τ is the performance guarantee of the best approximation algorithm for $1 \,|\, r_j \,|\, L_{\max}$. Hence the approx-

imation algorithm of Zdrzalka has performance guarantee arbitrarily close to $\frac{3}{2}$. Nowicki [6] improved on this result by constructing a $(\frac{4}{3}+\varepsilon)$-approximation algorithm, where $\varepsilon > 0$ can be arbitrarily small.

In this note we apply the ideas of Hall and Shmoys [2] for $1 \mid r_j \mid L_{\max}$ and some of the ideas of Hall [1] for the flowshop problem, to design an approximation scheme for minimizing the latest delivery time under controllable processing times.

2 The approximation scheme

Clearly the total cost is a combination of two opposing objectives: On the one hand we want to minimize the total length and on the other hand we wish to minimize the total compression cost. Therefore, it seems logical to separate these costs when attacking the problem. It comes in hand to express the minimal compression costs as a function of the length of the schedule; to that end we define $C_{opt}(L)$ to be the minimal compression costs of a schedule of length L. Note that the minimal total cost, T^*, equals $\min_L(L + C_{opt}(L))$. We start with the following observation.

Observation 2.1 *The minimal compression costs of a schedule $C_{opt}(L)$ is nonincreasing in its length L.* □

Before giving a detailed description of the approximation scheme, we start with a global explanation of the ideas behind the scheme. Our approach consists of finding a schedule with nearly optimal costs given a fixed length. We construct an algorithm $A_\varepsilon(L)$ that, given a feasible schedule length L, produces a schedule of length at most $(1+\frac{\varepsilon}{3})L$ and costs at most $C_{opt}(L)$.

As introduced by Hall and Shmoys in [2] and used in various papers later on, we make use of a so called *outline scheme*. An outline scheme is a partition of the feasible schedules into sets. This partition is done in such a way that schedules in the same sets, share the same characteristics. The idea of the outline scheme is to generate a good schedule for each set and then to take the best among these generated schedules to be the approximate solution. For this idea to work, the following two conditions are necessary: First, the partition has a polynomial number of sets, and second, for each set, we are able to find a schedule that is nearly as good as the best schedule within that set, in polynomial time.

To satify the first condition, as one usually does in designing a PTAS, we distinguish between big jobs and small jobs. We call a job *big* if its shortened processing time is at least $\delta^2 L$, otherwise we call a job *small*. Note that, in contrary to the usual definition, this definition is schedule dependent.

Each set in the outline scheme is characterized by, what we call, a *skeleton*. Roughly speaking, this skeleton determines the approximate position of the big jobs in a schedule; an exact characterization of a skeleton is given in Section 2.2. Our algorithm $A_\varepsilon(L)$, which is based on building a good schedule for each skeleton, consists of two stages. Given a candidate length L, we first guess the approximate position of the big jobs in an optimal schedule by enumerating all possible skeletons. In the second stage, we construct a schedule for each skeleton by fitting in the small

jobs. The best among all those schedules will be the output of our approximation algorithm.

Below we describe the various steps of our approximation scheme in detail: First, in Section 2.1, we indicate for which lengths L we construct an approximate solution; the characteristics of a skeleton are described in Section 2.2; in Section 2.3 we show how we have incorporated the small jobs to obtain good schedules. Finally, in Section 2.4, we show that our algorithm indeed produces a near-optimal solution, i.e. a solution with cost at most $(1+\varepsilon)T^*$.

2.1 Fixing the length of the schedule

As we want our algorithm to have polynomial running time, we only evaluate a constant number of different lengths L. In order to determine suitable lengths, we start by computing natural lower and upper bounds on the length of a schedule. By means of the approximation scheme of Hall and Shmoys [2], we get a lower bound on the length of a schedule by computing an approximate schedule in case all jobs are compressed upon their maximal compression u_j. We choose the parameters in the approximation scheme of Hall and Shmoys such that the approximate length of this schedule, which we denote by $L_0(\varepsilon)$, is at most $(1+\frac{\varepsilon}{3})$ times the length of an optimal schedule in which all jobs are maximaly compressed.

Analogously, we determine a natural upper bound on the length of a schedule by finding an approximate schedule for the problem in which no job is compressed. The approximate length of this schedule is denoted by $L_\infty(\varepsilon)$, where $L_\infty(\varepsilon)$ is less than $(1+\frac{\varepsilon}{3})$ times the length of an optimal schedule in which no job is compressed. Clearly, we only consider schedules with length between $L_0(\varepsilon)$ and $L_\infty(\varepsilon)$.

We would like subsequent lengths to differ at most a multiplicative factor of $1+\frac{\varepsilon}{3}$. For this purpose the range of $[L_0(\varepsilon), L_\infty(\varepsilon)]$ could be too wide, therefore we need to construct better lower and upper bounds.

The work of Zdrzalka [9] and Nowicki [6] gives us good lower and upper bounds on the total cost T^* of an optimal schedule. Consider a $\frac{3}{2}$-approximation algorithm for the problem, obtained by the approach of Nowicki. Let $T(\sigma_{\mathcal{N}})$ be the cost of schedule $\sigma_{\mathcal{N}}$ produced by this algorithm. Clearly, we do not execute $A_\varepsilon(L)$ for lengths L with $L > T(\sigma_{\mathcal{N}})$. Furthermore, in case $L < \frac{4}{9}\varepsilon T(\sigma_{\mathcal{N}})$, that is, $L < \frac{2}{3}\varepsilon T^*$, we also do not execute $A_\varepsilon(L)$.

Concluding: We execute algorithm $A_\varepsilon(L)$ for a constant number of lengths L, starting with $\max(L_0(\varepsilon), \frac{4}{9}\varepsilon T(\sigma_{\mathcal{N}}))$, increasing the length every time with a factor $(1+\frac{\varepsilon}{3})$ until the value exceeds $\min(L_\infty(\varepsilon), T(\sigma_{\mathcal{N}}))$. The structure of our scheme is as follows.

Scheme

INPUT: A number $\varepsilon < 1$.

Compute schedules σ_0 and σ_∞ with lengths $L_0(\varepsilon)$ and $L_\infty(\varepsilon)$ respectively, by means of the approximation scheme of Hall and Shmoys.
Construct a schedule $\sigma_{\mathcal{N}}$, with cost $T(\sigma_{\mathcal{N}})$, by a $\frac{3}{2}$-approximation algorithm obtained by the approach of Nowicki.

$L := \max(L_0(\varepsilon), \frac{4}{9}\varepsilon T(\sigma_{\mathcal{N}}))$;
WHILE $L < \min(L_\infty(\varepsilon), T(\sigma_{\mathcal{N}}))$ DO
    Construct an approximate schedule σ by algorithm $A_\varepsilon(L)$;
    $L := L(1 + \frac{\varepsilon}{3})$
END WHILE
OUTPUT: A schedule σ for which the total cost $T(\sigma)$ is minimal among all constructed schedules.

The algorithm $A_\varepsilon(L)$ consists of two stages, which are explained in the next two sections.

2.2 Stage 1: Characterising the skeleton of a schedule

Following the idea of the outline scheme, the first stage of $A_\varepsilon(L)$ consists of grouping together schedules with the same skeleton. We first characterize such a skeleton, after which we enumerate all feasible skeletons in order to build one good schedule from each of these skeletons in the second stage.

Given a candidate length L and a constant ε, we divide the interval $[0, L)$ in $\frac{1}{\delta}$ intervals I_i, $i = 1, \ldots, \frac{1}{\delta}$, of equal length, i.e. $I_i = [(i-1)L\delta, iL\delta)$, where $\delta = \frac{1}{18}\varepsilon$. For each interval I_i, we would like to know which jobs are started in this interval and the amount of time that they occupy the machine. As in e.g. Hall & Shmoys [3] and Hall [1], we do not determine the corresponding starting interval for all jobs, but only for the big jobs. The only difference between our problem and for example the flowshop problem studied by Hall, is that, due to the possibility of compressing, we do not know beforehand which jobs are big and which jobs are small. However, we can easily overcome this difference: a straightforward extension of the approach in Hall and Shmoys [3], leads to a PTAS.

We characterize a skeleton by the following:

- For each interval I_i, $i = 1, \ldots, \frac{1}{\delta}$, we specify a set of big jobs B_i. The cardinality of each set B_i is at most $\frac{1}{\delta}$ and $B_i \cap B_l = \emptyset$ for all i and l.
- For each job J_j in a set B_i, we specify its approximate shortened processing time $\hat{a}_j$; $\hat{a}_j$ is between $\delta^2 L$ and L, and is a multiple of $\delta^3 L$.
- For each interval I_i, $i = 1, \ldots, \frac{1}{\delta}$, we specify the approximate total shortened processing time $\hat{A}_i$ of the small jobs in the interval I_i; $\hat{A}_i$ is a multiple of $\delta^2 L$.

We can represent a skeleton by a vector $\bar{y}$, where $y_i = (B_i, \{\hat{a}_j | J_j \in B_i\}, \hat{A}_i)$. Although the number of different skeletons is polynomial (see the proof of Lemma 2.2 in Section 2.4), we restrict our attention to those skeletons for which there possibly exists a feasible schedule, i.e. so-called *feasible skeletons* that satisfy the following conditions.

- Every big job can be compressed up to $\hat{a}_j$, that is, for every job J_j in a set B_i:

$$p_j - u_j \leq \hat{a}_j \leq \left\lceil \frac{p_j}{\delta^3 L} \right\rceil \delta^3 L.$$

- Every big job is assigned to a compatible starting interval, i.e. for every job $J_j \in B_i$:

$$r_j < i\delta L \text{ and } (i-1)\delta L + \hat{a}_j + q_j \leq \left\lceil \frac{1}{\delta^3} \right\rceil \delta^3 L.$$

- No interval is overloaded, i.e. for every interval I_i and for all $1 \leq l \leq i$

$$\sum_{k=l}^{i} (\hat{A}_k + \sum_{j \in B_k} \hat{a}_j) - \max_{j \in B_i} \hat{a}_j \leq (i-l+1)(\delta L + 2\delta^2 L),$$

the additional $2\delta^2 L$ is due to the rounding of the shortened processing times.

Our approach consists in enumerating all feasible skeletons. In the next section we explain how to transform a feasible skeleton $\bar{y}$ into a feasible schedule $\sigma(\bar{y})$.

2.3 Stage 2: Incorporating the small jobs

Given a feasible skeleton $\bar{y}$, we know the approximate total amount of shortened processing time of the small jobs for each interval I_i and both the starting interval and the approximate shortened processing time of the big jobs. Since we have rounded the shortened processing times, we need to enlarge the intervals I_i. We therefore define intervals $\hat{I}_i$, $i = 1, \ldots, \frac{1}{\delta}$, where $\hat{I}_i = [(i-1)(\delta L + 3\delta^2 L), i(\delta L + 3\delta^2 L))$. We now determine the starting interval $\hat{I}_i$ and the shortened processing time for the small jobs. Analogous to Hall [1], we determine this data by means of a linear program.

To that end, we define decision variables a_{ij}, where a_{ij} represents the shortened processing time of job J_j in the interval $\hat{I}_i$. The set of small jobs is denoted by S. In fact our linear program assigns different pieces of the same job to different intervals, which corresponds to the construction of a preempted schedule. The first two inequalities in the LP-formulating below, express the bounds on the amount of compression x_j. Equalities three and four impose natural constraints on the intervals each (piece of) job is processed in, whereas inequalities five and six bound the total amount of shortened processing time for each machine and each job, respectively. The goal is of course to minimize the total compression cost.

LP

$$\begin{array}{lrcll}
\min & \sum_j c_j(p_j - \sum_i a_{ij}) & & & \\
\text{s.t.} & p_j - \sum_i a_{ij} & \geq & 0 & \forall j \in S \\
 & p_j - \sum_i a_{ij} & \leq & u_j & \forall j \in S \\
 & a_{ij} & = & 0 \quad \text{if } r_j \geq i\delta L & \forall i = 1, \ldots, \frac{1}{\delta}, \forall j \in S \\
 & a_{ij} & = & 0 \quad \text{if } L - q_j < (i-1)\delta L & \forall i = 1, \ldots, \frac{1}{\delta}, \forall j \in S \\
 & \sum_j a_{ij} & \leq & \hat{A}_i & \forall i = 1, \ldots, \frac{1}{\delta} \\
 & \sum_i a_{ij} & \leq & \delta^2 L & \forall j \in S \\
 & a_{ij} & \geq & 0 & \forall i = 1, \ldots, \frac{1}{\delta}, \forall j \in S
\end{array}$$

The LP may not give a solution, in this case it is clear that there is no schedule corresponding to the skeleton $\bar{y}$. Otherwise, we construct a feasible schedule as follows.

Instead of assigning different pieces of a small job to different intervals, as the LP-solution suggests, we assign the job as a whole, that is, the total shortened processing time $\sum_i a_{ij}$ as determined by the LP, to a single interval $\hat{I}_i$. We recursively determine the jobs that have starting interval $\hat{I}_1, \hat{I}_2, \ldots \hat{I}_{\frac{1}{\delta}}$. In each step i, we first compute R_i, which denotes the subset of small jobs that have not been assigned to $\hat{I}_1, \hat{I}_2, \ldots \hat{I}_{i-1}$ and for which there is a $l \leq i$ with $a_{lj} > 0$. Then, we order the jobs in R_i in increasing order of their delivery time. Finally, we assign the jobs in R_i one by one to $\hat{I}_i$ until the total shortened processing time exceeds $\hat{A}_i$.

Given the starting intervals of the jobs, we order the jobs in each interval $\hat{I}_i$ as follows. First we schedule the small jobs (in arbitrary order), then we schedule the big jobs in order of nondecreasing shortened processing time. We only allow idle time between two jobs with different starting intervals. In order to obtain a feasible schedule, that is, to ensure that each job is started at or after its release time, we introduce an idle interval with length δL at the beginning of the schedule. Hence the intervals $\hat{I}_i$ are shifted by δL time units. We call the schedule constructed above $\sigma(\bar{y})$. It is easy to check that $\sigma(\bar{y})$ is a feasible schedule.

Summarizing: the structure of our algorithm $A_\varepsilon(L)$ is as follows.

Algorithm $A_\varepsilon(L)$

INPUT: A number $\varepsilon < 1$ and an integer L.

$\delta := \frac{1}{18}\varepsilon$;
Divide the interval $[0, L)$ into $\frac{1}{\delta}$ intervals of equal length;
Enumerate all feasible skeletons $\bar{y}$;

FOR each feasible skeleton $\bar{y}$ DO
    Compute the compression cost $C_B(\bar{y})$ for the big jobs in $\bar{y}$
    Solve the LP;
    IF the LP has a solution with value $C_S(\bar{y})$
        THEN construct a schedule $\sigma(\bar{y})$ with length at most
        $(1 + \frac{\varepsilon}{3})L$ and compression cost $C_B(\bar{y}) + C_S(\bar{y})$.
END FOR

OUTPUT: A schedule σ with total minimal cost among all constructed schedules $\sigma(\bar{y})$.

2.4 The analysis

We start this section by showing that $A_\varepsilon(L)$ runs in polynomial time. Then, we conclude that, since the number of executions of $A_\varepsilon(L)$ is constant, as stated in Lemma 2.3, our scheme has polynomial running time. Finally, by means of Lemmas 2.4 and 2.5, we prove that our scheme produces a near-optimal solution.

Lemma 2.2 *Algorithm $A_\varepsilon(L)$ runs in polynomial time.*

Proof. Since the LP described in Section 2.3 clearly runs in polynomial time and the procedure to construct $\sigma(\bar{y})$ is also polynomial, the key factor is the number of different skeletons. If the latter is polynomial, then so is $A_\varepsilon(L)$.

As the number of big jobs per interval is at most $\frac{1}{\delta}$, we have at most $n^{\frac{1}{\delta}}$ different B_i's. For each job J_j we have at most $\frac{1}{\delta^3}$ choices for its approximate shortened processing time $\hat{a}_j$; hence, there are at most $\frac{1}{\delta^3}^{\frac{1}{\delta}}$ different sets $\{\hat{a}_j | J_j \in B_i\}$. Finally, there are $\frac{1}{\delta}$ different choices for $\hat{A}_j$. Concluding: the number of different skeletons is at most

$$\left((n\frac{1}{\delta^3})^{\frac{1}{\delta}}\frac{1}{\delta}\right)^{\frac{1}{\delta}},$$

and therefore $A_\varepsilon(L)$ runs in polynomial time. □

Lemma 2.3 *The number of times $A_\varepsilon(L)$ is executed is a constant that depends on ε.*

Proof. Let κ be the number of times we execute algorithm $A_\varepsilon(L)$ and let L_{first} and L_{last} be the first and last length, respectively, for which algorithm $A_\varepsilon(L)$ is executed. Clearly, $L_{first} \geq \frac{2}{3}\varepsilon T(\sigma_{\mathcal{N}})$ and $L_{last} = (1+\frac{\varepsilon}{3})^{\kappa-1}L_{first} < T(\sigma_{\mathcal{N}})$. Hence,

$$(1+\frac{\varepsilon}{3})^{\kappa-1}L_{first} < T(\sigma_{\mathcal{N}}) \leq \frac{3}{2\varepsilon}L_{first},$$

that is,

$$\kappa < \frac{\log(\frac{3}{2\varepsilon})}{\log(1+\frac{\varepsilon}{3})} + 1.$$ □

From the previous two lemmas it follows that our scheme runs in polynomial time. It now remains to prove that σ_ε, the output of our scheme, has value at most $(1+\varepsilon)T^*$. To that end, we first prove that algorithm $A_\varepsilon(L)$ outputs a near-optimal schedule.

Lemma 2.4 *The algorithm $A_\varepsilon(L)$ produces a schedule with length at most $(1+\frac{\varepsilon}{3})L$ and cost at most $C_{opt}(L)$.*

Proof. Let σ be a schedule with length at most L and cost at most $C_{opt}(L)$. We divide $[0, L)$ in $\frac{1}{\delta}$ intervals I_i of equal length and we define S_i to be the set of small jobs that start in I_i. Next, we construct a skeleton $\bar{y}$, with $y_i = (B_i, \{\hat{a}_j | J_j \in B_i\}, \hat{A}_i)$, where

- B_i is defined to be the set of big jobs that start in I_i;
- for each job $J_j \in B_i$, we define its approximate shortened processing time $\hat{a}_j$ to be $\lceil \frac{a_j}{\delta^3 L} \rceil \delta^3 L$;
- $\hat{A}_i$ is defined to be equal to $\left\lceil (\sum_{J_j \in S_i} a_j)/(\delta^2 L) \right\rceil \delta^2 L$.

After first enlarging each interval I_i by a multiplicative factor of $1+\delta$ (i.e. an absolute increase of δL), to compensate the rounding of the big jobs, and then enlarging each I_i by δL, to compensate for the rounding of the small jobs, it is clear that every big job can still be started within its assigned interval.

Given the skeleton $\bar{y}$, we construct a feasible schedule, $\sigma_\varepsilon(\bar{y})$, as described in Section 2.3. Thanks to the additional $\delta^2 L$ units of space in each interval $\hat{I}_i$, we can also cope with an additional small job that is possibly assigned to $\hat{I}_i$. Furthermore, since no big job is compressed more than in the original schedule (we have rounded up the shortened processing times) and the small jobs are compressed in such a way that the compression cost are minimized, the compression cost of $\sigma_\varepsilon(\bar{y})$ is at most $C_{opt}(L)$.

Let us now compute the length of $\sigma_\varepsilon(\bar{y})$. We have already argued that no interval $\hat{I}_i$ is overloaded, that is, the jobs assigned to $\hat{I}_i$ can actually be started in $\hat{I}_i$. Because our algorithm has shifted all jobs δL units to the right, no job is started before its release time. But a job might complete after L. Let us reason how much this additional delay might be. Consider a big job in $\sigma_\varepsilon(\bar{y})$ that starts at a time $\hat{t}_j \in \hat{I}_i$. In σ this job starts at time $t_j \in I_i$ i.e. $t_j \geq (i-1)\delta L$. As $t_j + a_j + q_j \leq L$,

$$\begin{aligned}
\hat{t}_j + \hat{a}_j + q_j &\leq \delta L + i(\delta L + 3\delta^2 L) + \hat{a}_j + q_j \\
&\leq 2\delta L + 3i\delta^2 L + t_j + (1+\delta)a_j + q_j \\
&\leq 5\delta L + (1+\delta)L \\
&\leq (1+6\delta)L \\
&= (1+\frac{\varepsilon}{3})L.
\end{aligned}$$

Hence, the delivery completion time of each big job is at most $(1+\frac{\varepsilon}{3})L$. A similar analysis can be made for each small job. Concluding: $L(\sigma_\varepsilon(\bar{y})) \leq (1+6\delta)L = (1+\frac{\varepsilon}{3})L$ and $C(\sigma_\varepsilon(\bar{y})) \leq C_{opt}(L)$. □

Lemma 2.5 *The scheme proposed in the previous sections computes a solution with value at most* $(1+\varepsilon)T^*$.

Proof. Let σ^* be an optimal schedule. We now distinguish two cases.

In case (i), the length $L(\sigma^*)$ of schedule σ^* is less than $\frac{2}{3}\varepsilon T^*$. We know that the algorithm $A_\varepsilon(\frac{2}{3}T^*)$ outputs a schedule σ_1 with total cost at most $(1+\frac{\varepsilon}{3})\frac{2}{3}\varepsilon T^* + C_{opt}(L(\sigma^*)) \leq \varepsilon T^* + T^* = (1+\varepsilon)T^*$. This settles the first case.

In case (ii), the length $L(\sigma^*)$ of schedule σ^* is at least $\frac{2}{3}\varepsilon T^*$. Then there exists an integer $k \geq 0$ such that $(1+\frac{\varepsilon}{3})^{k-1} L_{first} \leq L(\sigma^*) \leq (1+\frac{\varepsilon}{3})^k L_{first}$, where L_{first} is defined as in Lemma 2.3. Our scheme computes $A_\varepsilon((1+\frac{\varepsilon}{3})^k L_{first})$, which delivers a solution σ_2 with length at most $(1+\frac{\varepsilon}{3})^{k+1} L_{first} \leq (1+\frac{\varepsilon}{3})^2 L(\sigma^*) < (1+\varepsilon)L(\sigma^*)$ and cost at most $C_{opt}(L(\sigma^*))$.

To conclude, in either case our approximation algorithm outputs a schedule with cost at most $(1+\varepsilon)T^*$. □

Our final theorem summarizes the main result of this paper.

Theorem 2.6 *The single-machine problem with release dates, delivery times and compressible processing times with objective to minimize the maximal job delivery completion time possesses a PTAS.* □

References

[1] L.A. HALL. Approximability of flow shop scheduling. *Mathematical Programming 82*, 1998, 175–190.

[2] L.A. HALL AND D.B. SHMOYS, Approximation schemes for constrained scheduling problems, in *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, 1989, 134–140.

[3] L.A. HALL AND D.B. SHMOYS. Jackson's rule for single-machine scheduling: Making a good heuristic better. *Mathematics of Operations Research 17*, 1992, 22–35.

[4] D.S. HOCHBAUM AND D.B. SHMOYS. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *Journal of the ACM 34*, 1987, 144–162.

[5] J.K. LENSTRA, A.H.G. RINNOOY KAN, AND P. BRUCKER. Complexity of machine scheduling problems. *Annals of Discrete Mathematics 1*, 1977, 343–362.

[6] E. NOWICKI. An approximation algorithm for a single-machine scheduling problem with release times, delivery times and controllable processing times. *European Journal on Operations Research 72*, 1994, 74–81.

[7] C.N. POTTS. Analysis of a heuristic for one machine sequencing with release dates and delivery times. *Operations Research 28*, 1980, 1436–1441.

[8] L. SCHRAGE. Obtaining optimal solutions to resource constrained network scheduling problems. Unpublished manuscript, 1971.

[9] S. ZDRZALKA. Scheduling jobs on a single machine with release dates, delivery times and controllable processing times: Worst-case analysis. *Operations Research Letters 10*, 1991, 519–523.

Received May, 2002

Acta Cybernetica 15 (2002) 379–416.

An Arithmetic Theory of Consistency Enforcement

Sebastian Link* and Klaus-Dieter Schewe*

Abstract

Consistency enforcement starts from a given program specification S and a static invariant $\mathcal{I}$ and aims to replace S by a slightly modified program specification $S_{\mathcal{I}}$ that is provably consistent with respect to $\mathcal{I}$. One formalization which suggests itself is to define $S_{\mathcal{I}}$ as the greatest consistent specialization of S with respect to $\mathcal{I}$, where specialization is a partial order on semantic equivalence classes of program specifications.

In this paper we present such a theory on the basis of arithmetic logic. We show that with mild technical restrictions and mild restrictions concerning recursive program specifications it is possible to obtain the greatest consistent specialization gradually and independently from the order of given invariants as well as by replacing basic commands by their respective greatest consistent specialization. Furthermore, this approach allows to discuss computability and decidability aspects for the first time.

1 Introduction

In order to capture the semantics of a system, almost all approaches to formal specification provide at least static invariants. Then the problem is to guarantee consistency. For a program specification S and an invariant $\mathcal{I}$ this means that every execution of S starting in a state that satisfies $\mathcal{I}$ should always lead to a state satisfying $\mathcal{I}$, too. This is usually relaxed so that only terminating executions of S are considered, in which case the problems of termination and of consistency can be handled separately.

If program semantics is expressed axiomatically by the use of predicate transformers leading to weakest (liberal) preconditions, then consistency leads to the well known proof obligation $\mathcal{I} \Rightarrow wlp(S)(\mathcal{I})$. Verification of such proof obligations can then be a very hard task.

As an alternative consistency enforcement has been considered. In particular, in the field of databases, where the complexity of the invariants – usually called integrity constraints in this context [9] – is much higher than the complexity of the programs themselves, the trigger approach has become very popular, but it can be shown that triggers cannot solve the problem in general [7].

*Massey University, Department of Information Systems, Private Bag 11222, Palmerston North, NZ, E-mail: [s.link|k.d.schewe]@massey.ac.nz

Another approach considers *greatest consistent specializations* (GCSs) [6, 8]. Here the goal is to replace a given program specification S and a given static invariant $\mathcal{I}$ by a slightly modified program specification $S_{\mathcal{I}}$ that is provably consistent with respect to $\mathcal{I}$. The modification should guarantee that "effects" of the original S are preserved within $S_{\mathcal{I}}$. For this the approach considers the specialization order on semantic equivalence classes of program specifications. The existing theory is based on infinitary logic $\mathcal{L}^{\omega}_{\infty\omega}$.

In order to shift the GCS approach from the purely theoretical framework ([6]) to an applicable theory we have to investigate computability of GCSs and decidability of preconditions that must be built. For these purposes it is preferable to obtain a tight connection with classical recursion theory [1]. Therefore, we will replace the underlying logic of [6] by first-order arithmetic logic. The paper will introduce a new theory of consistency enforcement based on this logic with almost all results from [6] carrying over in a modified form. On this basis, effectivity issues can be investigated for the first time.

We start in Section 2 with a brief review of arithmetic logic. Then we show the existence of predicate transformers with respect to this logic. In particular, relational program semantics becomes equivalent to predicate transformer semantics provided we guarantee the property of universal conjunctivity and the pairing condition. We even show in Section 3 that recursion theory can be extended to the arithmetic case, at least, if we are restricted to certain WHILE-loops.

With this background we can show that the GCS approach carries over to arithmetic logic. This will be done in Section 4. Many of the proofs in [6] only require slight changes. Computability cannot be guaranteed in general, since the building of least fixpoints requires to test for semantic equivalence, which is undecidable. For the case of FOR-loops, however, GCSs are computable. This will be shown in Section 5. Furthermore, we show how effective GCSs can be computed.

We argue that at least for one application field, i.e. databases as already mentioned, the restrictions are tolerable. For the general case some other pragmatic solutions must be applied [5]. We conclude with a short summary and outlook.

Due to the compact representations in this paper we recommend reading [3] for details.

2 Arithmetic Logic and Programming Semantics

Our study is based on first-order arithmetic logic [1, Ch.7], i.e. our logical language contains just the function symbols 0, s, $+$ and $*$ of arity 0, 1, 2 and 2. The informal meaning is as usual: the constant 0, the succesor function, addition and multiplication. By convenience $+$ and $*$ are written as infix operators. The only predicate symbol is the equality symbol $=$. Variables in our language will be $x_1, x_2, x_3, \ldots$.

We use the notation $\mathbb{T}$ for the set of terms and $\mathbb{F}$ for the set of formulae. In addition, let V denote the set of variables. We allow all standard abbreviations including formulae *true* and *false*.

Semantically, we fix a structure with domain $\mathbb{N}$, the set of non-negative integers.

Then 0, s, $+$, $*$ and $=$ are interpreted in the usual way. For an interpretation it is then sufficient to consider a function $\sigma : V \to \mathbb{N}$. By the coincidence theorem it is even sufficient to be given the values $\sigma(x_i)$ for the free variables x_i in a term or a formula. In particular, we may always write σ as a k-tuple, if the number of free variables is k.

Finally, a k-ary relation $R \subseteq \mathbb{N}^k$ is called *arithmetical* iff it can be represented by a formula $Q \in \mathbb{F}$ in arithmetic logic (with free variables $x_1, \ldots, x_k$), i.e. $(a_1, \ldots, a_k) \in R$ holds iff $\models_\sigma Q$ holds for the interpretation defined by $\sigma(x_i) = a_i$ $(i = 1, \ldots, k)$.

2.1 Predicate Transformers in Arithmetic Logic

In accordance with the existing theory on consistency enforcement in [6] each finite subset $X \subseteq V$ is called a *state space*. Each function $\sigma : X \to \mathbb{N}$ is called a *state* on X. Equivalently, a state is always representable by a k-tuple. For a fixed X let Σ $(= \Sigma(X))$ denote the *set of all states over* X.

A formula $\varphi \in \mathbb{F}$ with free variables $fr(\varphi)$ in X is then called an *X-formula* or an *invariant* on X. In order to emphasize the variables we sometimes write $\varphi(\vec{x})$ with a vector $\vec{x}$ of the state variables involved.

Then any pair of formulae $(\Delta(S), \Sigma_0(S))$ with $2k$ and k free variables, respectively, may be considered as defining the *relational semantics* of a program specification S. For convenience assume the first k free variables in $\Delta(S)$ to coincide with the free variables of $\Sigma_0(S)$.

According to our notation we sometimes write $\Delta(S)(\vec{x}, \vec{y})$ and $\Sigma_0(S)(\vec{x})$. So $\Delta(S)$ can be interpreted by state pairs, whereas $\Sigma_0(S)$ allows an interpretation by states. We interpret (σ, τ) with $\models_{(\sigma,\tau)} \Delta(S)$ as an *execution* of S with start state σ and a final state τ. Similarly, a state σ satisfying $\Sigma_0(S)$ is considered as a start state for S, in which a non-terminating execution of S exists.

Note that the model of relational semantics comprises daemonic non-determinism, non-termination and partial undefinedness.

In order to come to an axiomatic semantics based on the introduced logic of arithmetic, we associate with S two *predicate transformers* $wlp(S)$ and $wp(S)$ – i.e., functions from (equivalence classes) of formulae to (equivalence classes) of formulae – with the standard informal meaning:

- $wlp(S)(\varphi)$ characterizes those initial states σ such that each terminating execution of S starting in σ results in a state τ satisfying φ.
- $wp(S)(\varphi)$ characterizes those initial states σ such that each execution of S starting in σ terminates and results in a state τ satisfying φ.

The notation $wlp(S)(\varphi)$ and $wp(S)(\varphi)$ corresponds to the usual *weakest (liberal) precondition* of S with respect to the postcondition φ. In order to save space we shall often use the notation $w(l)p(S)(\varphi)$ to refer to both predicate transformers at a time. If this occurs in an equivalence, then omitting everything in parentheses gives the wp-part, whereas omitting just the parentheses results in the wlp-part.

From our introduction of $\Delta(S)$ and $\Sigma_0(S)$ the following definition is straightforward.

Definition 1 The *predicate transformers* associated with a program specification S on a state space X are defined as

$$\begin{array}{rcll} wlp(S)(\varphi(\vec{x})) & \Leftrightarrow & \forall\vec{y}.\Delta(S)(\vec{x},\vec{y}) \Rightarrow \varphi(\vec{y}) & \text{and} \\ wp(S)(\varphi(\vec{x})) & \Leftrightarrow & (\forall\vec{y}.\Delta(S)(\vec{x},\vec{y}) \Rightarrow \varphi(\vec{y})) \wedge \neg\Sigma_0(S)(\vec{x}) & \end{array}$$

for arbitrary X-formulae φ. □

The next step is to show that predicate transformers satisfying some nice conditions are sufficient for the definition of program specifications S. The conditions are the *pairing condition* and a slightly modified *universal conjunctivity* property. This gives the equivalence between the relational and the predicate transformer semantics.

We use the standard notation $w(l)p(S)^*(\varphi) \Leftrightarrow \neg w(l)p(S)(\neg\varphi)$ and refer to $wlp(S)^*$ and $wp(S)^*$ as the *dual predicate transformers*.

Proposition 1 *The predicate transformers $w(l)p(S)$ satisfy the following conditions:*

$$\begin{array}{rcll} wp(S)(\varphi) & \Leftrightarrow & wlp(S)(\varphi) \wedge wp(S)(true) & \text{and} \\ wlp(S)(\forall\vec{y}.Q(\vec{y}) \Rightarrow \varphi(\vec{x},\vec{y})) & \Leftrightarrow & \forall\vec{y}.Q(\vec{y}) \Rightarrow wlp(S)(\varphi(\vec{x},\vec{y})) & \end{array}$$

Conversely, any pair of predicate transformers satisfying these two conditions defines $\Delta(S)(\vec{x},\vec{y}) \Leftrightarrow wlp(S)^*(\vec{x}=\vec{y})$ and $\Sigma_0(\vec{x}) \Leftrightarrow wp(S)^*(false)$.

Proof. We first show that $w(l)p(S)$ fulfil both conditions. Due to

$$\begin{array}{rcl} wp(S)(true) & \Leftrightarrow & (\forall\vec{y}.\Delta(S)(\vec{x},\vec{y}) \Rightarrow true) \wedge \neg\Sigma_0(S)(\vec{x}) \\ & \Leftrightarrow & \neg\Sigma_0(S)(\vec{x}) \end{array}$$

we receive the pairing condition

$$\begin{array}{rcl} wlp(S)(\varphi(\vec{x})) \wedge wp(S)(true) & \Leftrightarrow & (\forall\vec{y}.\Delta(S)(\vec{x},\vec{y}) \Rightarrow \varphi(\vec{y})) \wedge \neg\Sigma_0(S)(\vec{x}) \\ & \Leftrightarrow & wp(S)(\varphi(\vec{x})) \end{array}$$

The universal conjunctivity property follows from

$$\begin{array}{rcl} wlp(S)(\forall\vec{y}.Q(\vec{y}) \Rightarrow \varphi(\vec{x},\vec{y})) & \Leftrightarrow & \forall\vec{z}.\Delta(S)(\vec{x},\vec{z}) \Rightarrow \{\vec{x}/\vec{z}\}.(\forall\vec{y}.Q(\vec{y}) \Rightarrow \varphi(\vec{x},\vec{y})) \\ & \Leftrightarrow & \forall\vec{z}.\Delta(S)(\vec{x},\vec{z}) \Rightarrow (\forall\vec{y}.Q(\vec{y}) \Rightarrow \varphi(\vec{z},\vec{y})) \\ & \Leftrightarrow & \forall\vec{y}.\forall\vec{z}.(\Delta(S)(\vec{x},\vec{z}) \wedge Q(\vec{y}) \Rightarrow \varphi(\vec{z},\vec{y})) \\ & \Leftrightarrow & \forall\vec{y}.Q(\vec{y}) \Rightarrow \forall\vec{z}.(\Delta(S)(\vec{x},\vec{z}) \Rightarrow \varphi(\vec{z},\vec{y})) \\ & \Leftrightarrow & \forall\vec{y}.Q(\vec{y}) \Rightarrow \forall\vec{z}.(\Delta(S)(\vec{x},\vec{z}) \Rightarrow \{\vec{x}/\vec{z}\}.\varphi(\vec{x},\vec{y})) \\ & \Leftrightarrow & \forall\vec{y}.Q(\vec{y}) \Rightarrow wlp(S)(\varphi(\vec{x},\vec{y})) \end{array}$$

for the case that $\{\vec{y} \mid\!\models Q(\vec{y})\} \neq \emptyset$ holds. If this set is empty then $\neg Q(\vec{y})$ holds for all $\vec{y}$ and we have $wlp(S)(true) \Leftrightarrow true$ which is obviously valid.
Now, let f_{lp} and f_p be predicate transformers statisfying the pairing condition and the universal conjunctivity property. Then it remains to show $wlp(S) = f_{lp}(S)$ and $wp(S) = f_p(S)$. For an arbitrary $X-$formula φ we have

$$\models_\sigma \varphi(\vec{x}) \Leftrightarrow \models_\sigma \varphi'(\vec{x}) \quad \text{with} \quad \varphi'(\vec{x}) \Leftrightarrow \forall \vec{y}.\, (\vec{x} = \vec{y} \Rightarrow \varphi(\vec{y}))$$

Let σ be an arbitrary state with $\models_\sigma f_{lp}(S)(\varphi(\vec{x}))$. Then we compute

$$\begin{aligned}
\models_\sigma f_{lp}(S)(\varphi(\vec{x})) \quad &\Leftrightarrow \quad \models_\sigma f_{lp}(S)(\varphi'(\vec{x})) \\
&\Leftrightarrow \quad \models_\sigma f_{lp}(S)(\forall \vec{y}.\vec{x} = \vec{y} \Rightarrow \varphi(\vec{y})) \\
&\Leftrightarrow \quad \models_\sigma f_{lp}(S)(\forall \vec{y}.\neg\varphi(\vec{y}) \Rightarrow \vec{x} \neq \vec{y}) \\
&\Leftrightarrow \quad \models_\sigma \forall \vec{y}.\neg\varphi(\vec{y}) \Rightarrow f_{lp}(S)(\vec{x} \neq \vec{y}) \\
&\Leftrightarrow \quad \models_\sigma \forall \vec{y}.f_{lp}(S)^*(\vec{x} = \vec{y}) \Rightarrow \varphi(\vec{y}) \\
&\Leftrightarrow \quad \models_\sigma \forall \vec{y}.\Delta(S)(\vec{x}, \vec{y}) \Rightarrow \varphi(\vec{y}) \\
&\Leftrightarrow \quad \models_\sigma wlp(S)(\varphi(\vec{x})) \quad ,
\end{aligned}$$

therefore the asserted equivalence. Furthermore, we have

$$\begin{aligned}
wp(S)(\varphi) \quad &\Leftrightarrow \quad wlp(S)(\varphi) \wedge wp(S)(true) && \text{(pairing condition)} \\
&\Leftrightarrow \quad wlp(S)(\varphi) \wedge \neg\Sigma_0(S)(\vec{x}) && \text{(Def. } wp(S)(true)\text{)} \\
&\Leftrightarrow \quad wlp(S)(\varphi) \wedge \neg f_p(S)^*(false) && \text{(Def. } \Sigma_0(S)\text{)} \\
&\Leftrightarrow \quad f_{lp}(S)(\varphi) \wedge \neg f_p(S)^*(false) && (wlp(S) = f_{lp}(S)) \\
&\Leftrightarrow \quad f_{lp}(S)(\varphi) \wedge f_p(S)(true) && \text{(Def. } f_p(S)^*\text{)} \\
&\Leftrightarrow \quad f_p(S)(\varphi) \quad , && \text{(pairing condition)}
\end{aligned}$$

which completes the proof. □

The next result gives a normal form representation of the predicate transformer $wlp(S)$, which will be useful in many proofs.

Lemma 1 *It is always possible to write $wlp(S)(\varphi)$ in the form*

$$wlp(S)(\varphi(\vec{x})) \quad \Leftrightarrow \quad \forall \vec{z}.wlp(S)^*(\vec{x} = \vec{z}) \Rightarrow \varphi(\vec{z})$$

Proof.

Obviously, we have $\varphi(\vec{x}) \Leftrightarrow \forall \vec{z}.\vec{x} = \vec{z} \Rightarrow \varphi(\vec{z}) \Leftrightarrow \forall \vec{z}.\neg\varphi(\vec{z}) \Rightarrow \vec{x} \neq \vec{z}$. Then the lemma follows immediately by applying the universal conjunctivity property. □

2.2 Guarded Commands

We now introduce the familiar language of guarded commands [4]. We use *skip*, *fail*, *loop* and parallel assignment $x_{i_1} := t_{i_1} \| \ldots \| x_{i_k} := t_{i_k}$ with variables $x_{i_j} \in V$ and terms $t_{i_j} \in \mathbb{T}$ as basic commands. The informal meaning of the first three

in this list is to change nothing, to be completely undefined and to do only non-terminating executions, respectively.

Complex commands are constructed from sequences $S_1; S_2$, choices $S_1 \square S_2$, restricted choices $S_1 \boxtimes S_2$, unbounded choice $@x_j \bullet S$ and preconditioning $\mathcal{P} \to S$.

To define the semantics we simply have to define the predicate transformers. These are given as follows:

$$\begin{aligned}
w(l)p(skip)(\varphi) &\Leftrightarrow \varphi \\
w(l)p(fail)(\varphi) &\Leftrightarrow true \\
w(l)p(loop)(\varphi) &\Leftrightarrow false(\forall true) \\
w(l)p(x_{i_1} := t_{i_1} \| \ldots \| x_{i_k} := t_{i_k})(\varphi) &\Leftrightarrow \{x_{i_1}/t_{i_1}, \ldots, x_{i_k}/t_{i_k}\}.\varphi \\
w(l)p(S_1; S_2)(\varphi) &\Leftrightarrow w(l)p(S_1)(w(l)p(S_2)(\varphi)) \\
w(l)p(S_1 \square S_2)(\varphi) &\Leftrightarrow w(l)p(S_1)(\varphi) \wedge w(l)p(S_2)(\varphi) \\
w(l)p(S_1 \boxtimes S_2)(\varphi) &\Leftrightarrow w(l)p(S_1)(\varphi) \wedge (wp(S_1)^*(true) \vee w(l)p(S_2)(\varphi)) \\
w(l)p(@x_j \bullet S)(\varphi) &\Leftrightarrow \forall x_j.w(l)p(S)(\varphi) \\
w(l)p(\mathcal{P} \to S)(\varphi) &\Leftrightarrow \mathcal{P} \Rightarrow w(l)p(S)(\varphi)
\end{aligned}$$

Here $\{x_{i_1}/t_{i_1}, \ldots, x_{i_k}/t_{i_k}\}$ denotes the simultaneous substitution of the variables x_{i_j} by the terms t_{i_j}. We do not want to dispense with the *restricted choice*-operator $\boxtimes$ since it is needed to define IF S FI and DO S OD commands. For a deeper justification, please see [4]. Of course, we might always write $S_1 \square wp(S_1)(false) \to S_2$ instead of $S_1 \boxtimes S_2$. However, this violates the orthogonality property of guarded commands which we want to maintain.

It is easy to verify the pairing condition and the universal conjunctivity property for these predicate transformers.

We say that S is an *X-command* for some state space X iff $w(l)p(S)(\varphi) \Leftrightarrow \varphi$ hold for each Y-formulae φ, where $X \cap Y = \emptyset$, and X is minimal with this property.

3 Recursion

In the last section we introduced the language of guarded commands together with an axiomatic semantics expressed via predicate transformers in arithmetic logic. So far, this language covers straightline non-deterministic partial programs extended by unbounded choice. We would like to go a bit further and investigate recursive programs expressed as least fixpoints $\mu T.f(T)$ with respect to a suitable order $\preceq$. This order will be the standard Nelson-order [4].

Unfortunately, we are not able to carry over the very general recursion theory from [4]. We have to restrict ourselves to simple WHILE-loops, i.e. $f(T) = \mathcal{P} \to S; T \square \neg\mathcal{P} \to skip$, where the variable T does not occur within S. For convenience, we introduce command variables $T_1, T_2, \ldots$. Throughout this section, we will use $f(T)$ to denote simple WHILE-loops as above.

3.1 The Nelson-Order

The idea of the Nelson-order is that whenever $S_1 \preceq S_2$ holds, then each terminating execution of S_1 is preserved within S_2, but a terminating execution in S_2 may be "approximated" in S_1 by a non-terminating execution. This leads to the following definition.

Definition 2 The *Nelson-order* is defined by

$$S_1 \preceq S_2 \Leftrightarrow (wlp(S_2)(\varphi) \Rightarrow wlp(S_1)(\varphi)) \wedge (wp(S_1)(\varphi) \Rightarrow wp(S_2)(\varphi))$$

for all φ. □

Particularly, we are interested in chains $\{f^i(loop)\}_{i\in\mathbb{N}}$ with respect to $\preceq$. Therefore, we define next a Gödel numbering g of guarded commands, which extends the Gödel numbering of terms and formulae from [1, p.327f.]. Let h denote this Gödel numbering for our logic. Recall the following definition:

$$h(0) = 1, \quad h(x_i) = 3^i, \quad h(s(t)) = 2 \cdot 3^{h(t)}, \quad h(t_1 + t_2) = 4 \cdot 3^{h(t_1)} \cdot 5^{h(t_2)},$$
$$h(t_1 * t_2) = 8 \cdot 3^{h(t_1)} \cdot 5^{h(t_2)}, \; h(t_1 = t_2) = 16 \cdot 3^{h(t_1)} \cdot 5^{h(t_2)}, \; h(\neg\varphi) = 32 \cdot 3^{h(\varphi)},$$
$$h(\varphi_1 \Rightarrow \varphi_2) = 64 \cdot 3^{h(\varphi_1)} \cdot 5^{h(\varphi_2)} \quad \text{and} \quad h(\forall x_i.\, \varphi) = 2^{6+i} \cdot 3^{h(\varphi)}.$$

In the same way we define

$$g(fail) = 1, \quad g(loop) = 2, \quad g(skip) = 4,$$
$$g(x_{i_1} := t_{i_1} \| \ldots \| x_{i_k} := t_{i_k}) = 8 \cdot \prod_{j=1}^{k} prim(i_j)^{h(t_{i_j})},$$
$$g(S_1; S_2) = 16 \cdot 3^{g(S_1)} \cdot 5^{g(S_2)}, \quad g(S_1 \square S_2) = 32 \cdot 3^{g(S_1)} \cdot 5^{g(S_2)},$$
$$g(S_1 \boxtimes S_2) = 64 \cdot 3^{g(S_1)} \cdot 5^{g(S_2)},$$
$$g(\mathcal{P} \to S) = 128 \cdot 3^{h(\mathcal{P})} \cdot 5^{g(S)}, \quad \text{and} \quad g(@x_j \bullet S) = 256 \cdot 3^j \cdot 5^{g(S)}$$

with the primitive recursive function *prim* taking n to the n'th prime number.

First we show that with this Gödel numbering g we may express all formulae $w(l)p(f^i(loop))(\varphi)$ by two arithmetic predicate transformers.

Lemma 2 *Let $f(T) = \mathcal{P} \to S; T \square \neg\mathcal{P} \to skip$ such that T does not occur within S. Then for each $j \in \mathbb{N}$, there exist predicate transformers $\tau_1(j)$ and $\tau(j)$ on arithmetic predicates such that the following properties are satisfied:*

1. *for each arithmetic predicate $\varphi(\vec{x})$, the results of applying these predicate transformers are arithmetic predicates in i and x, say*

$$\chi_j^1(i, \vec{x}) = \tau_1(j)(\varphi(\vec{x})) \qquad \text{and} \qquad \chi_j^2(i, \vec{x}) = \tau(j)(\varphi(\vec{x}))$$

2. *for* $j = h(\varphi)$ *we obtain*

$$\forall \vec{x}.\forall i.\ (\chi_j^1(i,\vec{x}) \Leftrightarrow wlp(f^i(loop))(\varphi(\vec{x}))) \quad \text{and}$$
$$\forall \vec{x}.\forall i.\ (\chi_j^2(i,\vec{x}) \Leftrightarrow wp(f^i(loop))(\varphi(\vec{x})))$$

with $\vec{x} = x_{i_1},\ldots,x_{i_k}$.

Proof. It is sufficient to prove the lemma for the case of S not containing loops itself. In general, program specifications can only have finitely many loops, so we can find the claimed predicate transformers $\tau_l(j)$ and $\tau(j)$ for the innermost loop first. Here, the involved program specification S, say S_0, is non-recursive. Having proven the lemma for this case, we obtain valid predicate transformers $wlp(S_1)$ and $wp(S_1)$ for the innermost loop S_1 by Lemma 3. Hence, without loss of generality we can assume that S in $f(T) = \mathcal{P} \to S; T \square \neg\mathcal{P} \to skip$ is non-recursive.
For arbitrary program specifications T with $g(T) = i$ and arbitrary formulae $\varphi(\vec{x})$ with $h(\varphi) = j$ let us write $Q_1'(i,j,\vec{x}) = wlp(T)(\varphi(\vec{x}))$ and $Q_2'(i,j,\vec{x}) = wp(T)(\varphi(\vec{x}))$. If i,j are not Gödel numbers of programs or formulae, respectively, we may extend Q_1' and Q_2' arbitrarily. Let $prex(i,j)$ be the primitive recursive function that gives the exponent of the $j+1$-st prime number in the prime factorization of i. Then, we have

$$Q_1'(i,j,\vec{x}) = \begin{cases} true & ,prex(i,0) = 0 \\ true & ,prex(i,0) = 1 \\ h^{-1}(j) & ,prex(i,0) = 2 \\ \{x_{i_1}/h^{-1}(j_1),\ldots,x_{i_k}/h^{-1}(j_k)\}.h^{-1}(j) & ,prex(i,0) = 3 \\ & prex(i,i_l) = j_l \\ & \text{with } 1 \le l \le k \\ Q_1'(prex(i,1),Q_1'(prex(i,2),j,\vec{x}),\vec{x}) & ,prex(i,0) = 4 \\ Q_1'(prex(i,1),j,\vec{x}) \wedge Q_1'(prex(i,2),j,\vec{x}) & ,prex(i,0) = 5 \\ Q_1'(prex(i,1),j,\vec{x}) \wedge (Q_2'(prex(i,1),7,\vec{x}) & \\ \quad \Rightarrow Q_1'(prex(i,2),j,\vec{x})) & ,prex(i,0) = 6 \\ h^{-1}(prex(i,1)) \Rightarrow Q_1'(prex(i,2),j,\vec{x}) & ,prex(i,0) = 7 \\ \forall x_{prex(i,1)}.Q_1'(prex(i,2),j,\vec{x}) & ,prex(i,0) = 8 \end{cases}$$

We obtain a similar equation for $Q_2'(i,j,\vec{x})$ which does not depend on Q_1'. As this is a recursive definition, Q_1' is not an arithmetic predicate. Note, however, that if we fix i and j, i.e., the program specification T and the formula φ, we can turn the equation into a formula of arithmetic logic.
Let us now consider just the case $T = f^k(loop)$ for our fixed mapping f on program specifications. For $k = 0$ we have $wlp(loop)(\varphi(\vec{x})) \Leftrightarrow true$. Furthermore, we get $wlp(f^{k+1}(loop))(\varphi(\vec{x})) \Leftrightarrow ((\mathcal{P} \Rightarrow wlp(S)(wlp(f^k(loop))(\varphi(\vec{x})))) \wedge (\neg\mathcal{P} \Rightarrow \varphi(\vec{x})))$. Thus, we may define a primitve recursive function $\bar{g}$ with $\bar{g}(0) = g(loop)$ and

$$\bar{g}(k+1) = g(f^{k+1}(loop)) = 32 \cdot 3^{128} \cdot 3^{h(\mathcal{P})} \cdot 5^{16} \cdot 3^{g(S)} \cdot 5^{\bar{g}(k)} \cdot 5^{128} \cdot 3^{h(\neg\mathcal{P})} \cdot 5^4$$

such that

$$Q_1'(\bar{g}(k),j,\vec{x}) \quad = \quad wlp(f^k(loop))(\varphi(\vec{x}))$$

is satisfied. Now define an arithmetic formula $\bar{Q}(i,j,\vec{x})$ such that we have

$$\bar{Q}(h(\psi),h(\varphi),\vec{x}) \quad \Leftrightarrow \quad ((\mathcal{P} \Rightarrow wlp(S)(\psi)) \wedge (\neg\mathcal{P} \Rightarrow \varphi))$$

for arbitrary $\psi,\varphi \in \mathbb{F}$. As S is fixed and recursion-free we just take the right-hand side of the equivalence as the definition for $\bar{Q}(i,j,\vec{x})$ for Gödel numbers i,j of formulae and extend this to all i,j. If we take $Q_1(k,j,\vec{x}) = Q_1'(\bar{g}(k),j,\vec{x})$, we obtain (for $k > 0$)

$$Q_1(k,j,\vec{x}) \quad = \quad \bar{Q}(h(\psi),j,\vec{x})$$

with $\psi(\vec{x}) = wlp(f^{k-1}(loop))(\varphi(\vec{x}))$. Hence, also

$$\begin{array}{rcl} Q_1(0,j,\vec{x}) & = & true \quad \text{and} \\ Q_1(k+1,j,\vec{x}) & = & \bar{Q}(h(Q_1(k,j,\vec{x})),j,\vec{x}). \end{array}$$

Taking $\tau_l(j)(\varphi(\vec{x})) = \chi^1_j(k,\vec{x}) = Q_1(k,j,\vec{x})$ (for fixed j), this shows that $\chi^1_j(k,\vec{x})$ is arithmetic, as $\bar{Q}$ is arithmetic and arithmetic predicates are closed under primitive recursion. An analogous argument leads to arithmetic predicates $\chi^2_j(k,\vec{x}) = \tau(j)(\varphi(\vec{x}))$ for fixed j, thus proving the first part of the lemma. The equivalence in the second part follows immediately from the construction. □

With help of the arithmetic predicate transformers $\tau_l(j)$ and $\tau(j)$ from Lemma 2 we can now define a *limit operator* $S = \lim_{k\in\mathbb{N}} f^k(loop)$ via

$$\begin{array}{rcll} wlp(S)(\varphi(\vec{x})) & \Leftrightarrow & \forall k.\chi^1_{h(\varphi)}(k,\vec{x}) & \text{and} \\ wp(S)(\varphi(\vec{x})) & \Leftrightarrow & \exists k.\chi^2_{h(\varphi)}(k,\vec{x}) & . \end{array}$$

for $\chi^1_{h(\varphi)}(k,\vec{x}) = \tau_l(h(\varphi))(\varphi(\vec{x}))$ and $\chi^2_{h(\varphi)}(k,\vec{x}) = \tau(h(\varphi))(\varphi(\vec{x}))$.

Lemma 3 *The definition of $S = \lim_{i\in\mathbb{N}} f^i(loop)$ is sound.*

Proof. We first verify the universal conjunctivity property by direct calculation, namely

$$\begin{array}{rcl} wlp(S)(\forall\vec{z}.P(\vec{z}) \Rightarrow \varphi(\vec{x},\vec{z})) & \Leftrightarrow & \forall i.\chi^1_{h(\forall\vec{z}.P(\vec{z})\Rightarrow\varphi(\vec{x},\vec{z}))}(i,\vec{x}) \\ & \Leftrightarrow & \forall i.wlp(f^i(loop))(\forall\vec{z}.P(\vec{z}) \Rightarrow \varphi(\vec{x},\vec{z})) \\ & \Leftrightarrow & \forall i.\forall\vec{z}.P(\vec{z}) \Rightarrow wlp(f^i(loop))(\varphi(\vec{x},\vec{z}))) \\ & \Leftrightarrow & \forall\vec{z}.P(\vec{z}) \Rightarrow (\forall i.wlp(f^i(loop))(\varphi(\vec{x},\vec{z}))) \\ & \Leftrightarrow & \forall\vec{z}.P(\vec{z}) \Rightarrow \forall i.\chi^1_{h(\varphi)}(i,(\vec{x},\vec{z})) \\ & \Leftrightarrow & \forall\vec{z}.P(\vec{z}) \Rightarrow wlp(S)(\varphi(\vec{x},\vec{z})) \quad . \end{array}$$

For the second part of this Lemma, we first observe that

$$\begin{array}{rcl} wp(S)(\varphi(\vec{x})) & \Leftrightarrow & \exists i.\chi^2_{h(\varphi)}(i,\vec{x}) \\ & \Leftrightarrow & \exists i.wp(f^i(loop))(\varphi(\vec{x})) \\ & \Leftrightarrow & \exists i.wlp(f^i(loop))(\varphi(\vec{x})) \wedge wp(f^i(loop))(true) \end{array}$$

holds. In order to derive the pairing condition we verify both implications separately. Let us first show

$$wp(S)(\varphi) \quad \Rightarrow \quad wlp(S)(\varphi) \wedge wp(S)(true) \quad .$$

For a state σ with $\models_\sigma wp(S)(\varphi)$ it follows that $\models_\sigma wp(f^{i_0}(loop))(\varphi)$ holds, i.e. $\models_\sigma wlp(f^{i_0}(loop))(\varphi)$ and $\models_\sigma wp(f^{i_0}(loop))(true)$ for a particular $i_0 \in \mathbb{N}$. From $wp(S)(true) \Leftrightarrow \exists i.wp(f^i(loop))(true)$ we conclude $\models_\sigma wp(S)(true)$ and since $\{f^i(loop)\}_{i\in\mathbb{N}}$ is a chain it must be the case for every $i \in \mathbb{N}$ that either $f^i(loop) \preceq f^{i_0}(loop)$ or $f^{i_0}(loop) \preceq f^i(loop)$ holds which means either

$$\models_\sigma wlp(f^{i_0}(loop))(\varphi) \Rightarrow wlp(f^i(loop))(\varphi)$$

or

$$\models_\sigma wp(f^{i_0}(loop))(\varphi) \Rightarrow wlp(f^i(loop))(\varphi) \quad .$$

In every case, we have $\models_\sigma wlp(f^i(loop))(\varphi)$ for arbitrary $i \in \mathbb{N}$, therefore $\models_\sigma \forall i.wlp(f^i(loop))(\varphi)$, too and this is equivalent to $\models_\sigma wlp(S)(\varphi)$.
For the reverse direction

$$wlp(S)(\varphi) \wedge wp(S)(true) \quad \Rightarrow \quad wp(S)(\varphi)$$

we assume that $\models_\sigma \forall i.wlp(f^i(loop))(\varphi) \wedge \exists i.wp(f^i(loop))(true)$ holds. From this we derive $\models_\sigma wlp(f^{i_0}(loop))(\varphi) \wedge wp(f^{i_0}(loop))(true)$ for some $i_0 \in \mathbb{N}$, i.e. $\models_\sigma wp(f^{i_0}(loop))(\varphi)$ by the pairing condition of $f^{i_0}(loop)$. Finally, the assertion follows from $wp(S)(\varphi) \Leftrightarrow \exists i.wp(f^i(loop))(\varphi)$. □

3.2 Least Fixpoints

Now, we are going to show how to obtain the semantics for WHILE-loops. It is easy to see that the function $f(T) = \mathcal{P} \rightarrow S;T \square \neg\mathcal{P} \rightarrow skip$ on guarded commands is monotonic in the Nelson order [4]. Then an immediate consequence of the last lemma is the existence of a least upper bound, which is just given by the limit operator.

Lemma 4 *The chain* $\{f^i(loop) \mid i \in \mathbb{N}\}$ *has a least upper bound, namely* $\lim_{i\in\mathbb{N}} f^i(loop)$. □

Proof. We have already seen in the proof of Lemma 3 that

$$wlp(\lim_{i\in\mathbb{N}} f^i(loop))(\varphi) \quad \Leftrightarrow \quad \forall i.wlp(f^i(loop))(\varphi)$$

holds which means we receive $wlp(\lim_{i\in\mathbb{N}} f^i(loop))(\varphi) \Rightarrow wlp(f^k(loop))(\varphi)$ for all $k \in \mathbb{N}$. In addition, we have obtained

$$wp(\lim_{i\in\mathbb{N}} f^i(loop))(\varphi) \quad \Leftrightarrow \quad \exists i.wp(f^i(loop))(\varphi)$$

and because of that $wp(f^k(loop))(\varphi) \Rightarrow wp(\lim_{i\in\mathbb{N}} f^i(loop))(\varphi)$ for all $k \in \mathbb{N}$. Consequently, $\lim_{i\in\mathbb{N}} f^i(loop)$ is an upper bound of the chain $\{f^i(loop) \mid i \in \mathbb{N}\}$ with respect to the Nelson-order.
Now, let T be an arbitrary upper bound of $\{f^i(loop) \mid i \in \mathbb{N}\}$. Then we have to show $\lim_{i\in\mathbb{N}} f^i(loop) \preceq T$ but this follows immediately from

$$wlp(T)(\varphi) \quad \Rightarrow \quad wlp(f^i(loop))(\varphi) \text{ for all } i \in \mathbb{N} \quad \Leftrightarrow \quad wlp(\lim_{i\in\mathbb{N}} f^i(loop))(\varphi)$$

and

$$wp(\lim_{i\in\mathbb{N}} f^i(loop))(\varphi) \quad \Leftrightarrow \quad wp(f^i(loop))(\varphi) \text{ for some } i \in \mathbb{N} \quad \Rightarrow \quad wp(T)(\varphi)\ .$$

Thus, $\lim_{i\in\mathbb{N}} f^i(loop)$ is the least upper bound as asserted. □

In the following we use the notation $\mu T.f(T)$ to denote the least fixpoint of f provided it exists. We now restrict ourselves to WHILE-loops.

Proposition 2 *Let $f(T) = \mathcal{P} \to S; T \square \neg\mathcal{P} \to skip$. Then f has a least fixpoint with respect to $\preceq$, which is $\mu T.f(T) = \lim_{i\in\mathbb{N}} f^i(loop)$.*

Proof. First of all $\{f^i(loop) \mid i \in \mathbb{N}\}$ is a chain with respect to the Nelson-order since *loop* is a minimum and f is monotonic. Therefore, $\bar{S} = \lim_{i\in\mathbb{N}} f^i(loop)$ is the least upper bound according to Lemma 4. At this point we want to verify that $\bar{S}$ is a fixpoint with respect to f. Due to

$$\begin{aligned}
wlp(f(\bar{S}))(\varphi) \quad &\Leftrightarrow \quad (\mathcal{P} \Rightarrow wlp(T)(wlp(\bar{S})(\varphi))) \wedge (\neg\mathcal{P} \Rightarrow \varphi) \\
&\Leftrightarrow \quad (\mathcal{P} \Rightarrow wlp(T)(\forall i.i \in \mathbb{N} \Rightarrow wlp(f^i(loop))(\varphi))) \wedge (\neg\mathcal{P} \Rightarrow \varphi) \\
&\Leftrightarrow \quad (\mathcal{P} \Rightarrow (\forall i.i \in \mathbb{N} \Rightarrow wlp(T)(wlp(f^i(loop))(\varphi)))) \wedge (\neg\mathcal{P} \Rightarrow \varphi) \\
&\Leftrightarrow \quad (\forall i.i \in \mathbb{N} \Rightarrow (\mathcal{P} \Rightarrow wlp(T)(wlp(f^i(loop))(\varphi))) \wedge (\neg\mathcal{P} \Rightarrow \varphi) \\
&\Leftrightarrow \quad \forall i.i \in \mathbb{N} \Rightarrow (\mathcal{P} \Rightarrow wlp(T)(wlp(f^i(loop))(\varphi)) \wedge (\neg\mathcal{P} \Rightarrow \varphi)) \\
&\Leftrightarrow \quad \forall i.i \in \mathbb{N} \Rightarrow wlp(\mathcal{P} \to T; f^i(loop) \square \neg\mathcal{P} \to skip)(\varphi) \\
&\Leftrightarrow \quad \forall i.i \in \mathbb{N} \Rightarrow wlp(f(f^i(loop)))(\varphi) \\
&\Leftrightarrow \quad \forall i.i \in \mathbb{N} \Rightarrow wlp(f^i(loop))(\varphi) \\
&\Leftrightarrow \quad wlp(\lim_{i\in\mathbb{N}} f^i(loop))(\varphi) \\
&\Leftrightarrow \quad wlp(\bar{S})(\varphi) \quad ,
\end{aligned}$$

it remains to show $wp(f(\bar{S}))(true) \Leftrightarrow wp(\bar{S})(true)$. From the monotonicity of f it follows that $f(\bar{S})$ is a further upper bound of $\{f^i(loop) \mid i \in \mathbb{N}\}$ with respect to the Nelson-order, so we can conclude $\bar{S} \preceq f(\bar{S})$, especially

$$wp(\bar{S})(\varphi) \quad \Rightarrow \quad wp(f(\bar{S}))(\varphi)$$

Moreover, we receive

$$\begin{aligned}
wp(f(\bar{S}))(\varphi) &\Leftrightarrow (\mathcal{P} \Rightarrow wp(T)(wp(\bar{S})(\varphi))) \wedge (\neg\mathcal{P} \Rightarrow \varphi) \\
&\Leftrightarrow (\mathcal{P} \Rightarrow wp(T)(\exists i.i \in \mathbb{N} \wedge wp(f^i(loop))(\varphi))) \wedge (\neg\mathcal{P} \Rightarrow \varphi) \\
&\Rightarrow (\mathcal{P} \Rightarrow wp(T)(wp(f^i(loop))(\varphi))) \wedge (\neg\mathcal{P} \Rightarrow \varphi) \quad \text{for some } i \in \mathbb{N} \\
&\Leftrightarrow wp(\mathcal{P} \rightarrow T; f^i(loop) \Box \neg\mathcal{P} \rightarrow skip)(\varphi) \quad \text{for some } i \in \mathbb{N} \\
&\Leftrightarrow wp(\mathcal{P} \rightarrow T; f^i(loop) \Box \neg\mathcal{P} \rightarrow skip)(\varphi) \quad \text{for some } i \in \mathbb{N} \\
&\Leftrightarrow wp(f^{i+1}(loop))(\varphi) \quad \text{for some } i \in \mathbb{N} \quad ,
\end{aligned}$$

i.e. as to be shown

$$wp(f(\bar{S}))(\varphi) \quad \Rightarrow \quad \exists i.i \in \mathbb{N} \wedge wp(f^i(loop))(\varphi) \quad \Leftrightarrow \quad wp(\bar{S})(\varphi) \quad .$$

Let $\bar{T}$ be an arbitrary fixpoint with respect to f. Since *loop* is a minimum with respect to the Nelson-order we have $loop \preceq \bar{T}$. Applying the monotonicity of f with respect to $\preceq$ again we obtain $f^n(loop) \preceq f^n(\bar{T}) = \bar{T}$ for arbitrary $n \in \mathbb{N}$, so $\bar{T}$ is an upper bound of $\{f^i(loop) \mid i \in \mathbb{N}\}$ with respect to the Nelson-order. But $\bar{S}$ is the least upper bound, thus $\bar{S} \preceq \bar{T}$ holds. □

Finally, in order to support also nested loops, we extend the Gödel numbering g to command variables and fixpoint expression letting

$$g(T_j) = 512 \cdot 3^j \qquad \text{and} \qquad g(\mu T_j.f(T_j)) = 1024 \cdot 3^j \cdot 5^{g(f(T_j))} \; .$$

For the extension of Q_1' and Q_2' from the proof of Lemma 2 we then need a function $\ell(x, j, k)$, which associates with the Gödel number $x = g(f(T_j))$ the Gödel number $g(f^j(loop))$. We omit the details.

4 Greatest Consistent Specializations

Now the foundations are laid to develop the theory of consistency enforcement on top of first-order arithmetic logic.

4.1 Consistency and Specialization

First we have to define consistency and the specialization preorder. This can be done in complete analogy to the case in [6].

Definition 3 Let $\mathcal{I}$ be an invariant on the state space X. Let S and T be commands on the state spaces Z and Y, respectively, with $Z \subseteq Y \subseteq X$.

- S is *consistent* with respect to $\mathcal{I}$ iff $\mathcal{I} \Rightarrow wlp(S)(\mathcal{I})$ holds.
- T *specializes* S (notation: $T \sqsubseteq S$) iff $w(l)p(S)(\varphi) \Rightarrow w(l)p(T)(\varphi)$ holds for all Z-formulae φ. □

Due to the pairing condition it is sufficient to consider only $\varphi = true$ for the wp-part in the specialization definition. The wlp-part can also be simplified in the known way. The proof of the next proposition is shifted into Appendix A. The result will play an important role in the proof of Theorem 2.

Proposition 3 *Let S and T be commands on the state spaces X and Y, respectively, with $X \subseteq Y$. Then $wlp(S)(\varphi) \Rightarrow wlp(T)(\varphi)$ holds for all X-formulae iff*

$$\{\vec{z}/\vec{x}\}.wlp(T')(wlp(S)^*(\vec{x} = \vec{z}))$$

holds, where $\vec{z}$ is a disjoint copy of $\vec{x}$ and T' results from T by renaming each x_i into z_i. □

Next we introduce the central notion for consistency enforcement, the GCS.

Definition 4 Let S be a Y-command and $\mathcal{I}$ an invariant on X with $Y \subseteq X$. The *greatest consistent specialization* (GCS) of S with respect to $\mathcal{I}$ is an X-command $S_{\mathcal{I}}$ with $S_{\mathcal{I}} \sqsubseteq S$, such that $S_{\mathcal{I}}$ is consistent with respect to $\mathcal{I}$ and each consistent specialization $T \sqsubseteq S$ satisfies $T \sqsubseteq S_{\mathcal{I}}$. □

First we show the existence of GCSs and their uniqueness up to semantic equivalence. Furthermore, GCSs with respect to conjunctions can be built successively. In both cases, the proofs from [8, 6] carry over without significant changes. Nevertheless, we will give the proofs in Appendix B.

Proposition 4 *The GCS $S_{\mathcal{I}}$ of S with respect to $\mathcal{I}$ always exists and is unique up to semantic equivalence. We can always write*

$$S_{\mathcal{I}} = (\mathcal{I} \to (S; @\vec{z}' \bullet \vec{z} := \vec{z}'; \mathcal{I} \to skip)) \boxtimes (\neg\mathcal{I} \to (S; @\vec{z}' \bullet \vec{z} := \vec{z}'))\ ,$$

where $\vec{z}$ refers to the free variables in $\mathcal{I}$ not occurring in S.

Furthermore, for two invariants $\mathcal{I}$ and $\mathcal{J}$ we always obtain that $\mathcal{I} \wedge \mathcal{J} \to S_{\mathcal{I}\wedge\mathcal{J}}$ and $\mathcal{I} \wedge \mathcal{J} \to (S_{\mathcal{I}})_{\mathcal{J}}$ are semantically equivalent. □

The normal form of $S_{\mathcal{I}}$ of Proposition 4 should be read as follows. Whenever $\mathcal{I}$ holds, we execute S and permit arbitrary assignments to state variables that are not affected by S. Subsequently, we test whether $\mathcal{I}$ was indeed invariant under the execution of S and these assignments. For the case that $\mathcal{I}$ does not hold, we do not need to check $\mathcal{I}$ again. Using the normal form of Proposition 4, we may derive $wp(S_{\mathcal{I}})(true) \Leftrightarrow wp(S)(true)$ by direct computation. In fact, this is already obtainable from the definition of greatest consistent specializations. Anyway, this result allows us to concentrate on the predicate transformer $wlp(S)$.

4.2 An Upper Bound for GCSs

For practical applications the form of the GCS derived in Proposition 4 is almost worth nothing, since it involves testing the invariant after non-deterministic selection of arbitrary values. However, the form is useful in proofs.

A suitable form of the GCS should be built from GCSs of the basic commands involved in S. Let the result of such a naive syntactic replacement be denoted by $S'_{\mathcal{I}}$. In general, however, $S'_{\mathcal{I}}$ is not the GCS. It may not even be a specialization of S, or it may be a consistent specialization, but not the greatest one. An example for the latter case is $S = x := x - a; x := x + a$ with some constant $a \geq 1$ and $\mathcal{I} \equiv x \geq 1$.

We now formulate a technical condition which allows us to exclude this situation. Under this condition it will be possible to show that $S_{\mathcal{I}} \sqsubseteq S'_{\mathcal{I}}$ holds. The corresponding result will be called the *upper bound theorem.*

We need the notion of a *deterministic branch* S^+ of a command S, which requires $S^+ \sqsubseteq S$, $wp(S)^*(true) \Leftrightarrow wp(S^+)^*(true)$ and $wlp(S^+)^*(\varphi) \Rightarrow wp(S^+)(\varphi)$ to hold for all φ. Herein, the last condition expresses that S^+ is indeed deterministic, i.e., whenever $\models_{(\sigma,\tau)} \Delta(\vec{x},\vec{y})$ then $\models_\sigma \neg\Sigma_0(\vec{x})$ and whenever $\models_{(\sigma,\tau_1)} \Delta(\vec{x},\vec{y})$ and $\models_{(\sigma,\tau_2)} \Delta(\vec{x},\vec{y})$ hold then $\tau_1(\vec{x}) = \tau_2(\vec{x})$. Together, a deterministic branch S^+ of S is a deterministic specialization of S which comprises executions if and only if S does.

Furthermore, we need the notion of a *δ-constraint* for an X-command S. This is an invariant $\mathcal{J}$ on $X \cup X'$ with a disjoint copy X' of X, for which $\{\vec{x}'/\vec{x}\}.wlp(S')(\mathcal{J})$ holds, where S' results from S by renaming all x_i to x'_i. Thus, δ-constraints are exactly those formulae which are interpreted by state pairs and satisfied by a specification.

Finally, we write φ_σ for the characterizing formula of state σ.

Definition 5 Let $S = S_1; S_2$ be a Y-command such that S_i is a Y_i-command for $Y_i \subseteq Y$ $(i = 1, 2)$. Let $\mathcal{I}$ be some X-invariant with $Y \subseteq X$. Let $X - Y_1 = \{y_1, \ldots, y_m\}$, $Y_1 = \{x_1, \ldots, x_l\}$ and assume that $\{x'_1, \ldots, x'_l\}$ is a disjoint copy of Y_1 disjoint also from X. Then S is in *δ-$\mathcal{I}$-reduced form* iff for each deterministic branch S_1^+ of S_1 the following two conditions – with $\vec{x} = (x_1, \ldots, x_l)$, $\vec{x'} = (x'_1, \ldots, x'_l)$ – hold:

- For all states σ with $\models_\sigma \neg\mathcal{I}$ we have, if $\varphi_\sigma \Rightarrow \{\vec{x}/\vec{x'}\}.(\forall y_1 \ldots y_m.\mathcal{I})$ is a δ-constraint for S_1^+, then it is also a δ-constraint for $S_1^+ \,;\, S_2$.
- For all states σ with $\models_\sigma \mathcal{I}$ we have, if $\varphi_\sigma \Rightarrow \{\vec{x}/\vec{x'}\}.(\forall y_1 \ldots y_m.\neg\mathcal{I})$ is a δ-constraint for S_1^+, then it is also a δ-constraint for $S_1^+ \,;\, S_2$. □

Informally, δ-$\mathcal{I}$-reducedness is a property of sequences $S_1; S_2$ which rules out occurences of interim states that wrongly cause an enforcement within any branch of S_1 but which is not relevant for the entire specification. If we for instance look again at the example above, then the GCS of $S = x := x - a; x := x + a$ with respect to $\mathcal{I} \equiv x \geq 1$ is certainly *skip*, but $(x := x - a)_{\mathcal{I}} = (x = 0 \vee x > a) \rightarrow x := x - a$. A simple replacement of basic commands by their respective GCSs leads in this case to $(x = 0 \vee x > a) \rightarrow x := x - a; x := x + a$ which is just a proper specialization of *skip*. The reason for this is, that S is not in $\mathcal{I}$-reduced form.
Arbitrary programs S are called $\mathcal{I}$-reduced iff all occurences of sequences within S are δ-$\mathcal{I}$-reduced.

Definition 6 Let S be an Y-command and $\mathcal{I}$ some X-invariant with $Y \subseteq X$. S is called *$\mathcal{I}$-reduced* iff the following holds:

- If S is one of $fail$, $skip$, $loop$ or an assignment, then S is always $\mathcal{I}$-reduced.
- If $S = S_1; S_2$, then S is $\mathcal{I}$-reduced iff S_1 and S_2 are $\mathcal{I}$-reduced and S is δ-$\mathcal{I}$-reduced.
- If S is one of $\mathcal{P} \to T$, $@y \bullet T$, $S_1 \square S_2$ or $S_1 \boxtimes S_2$, then S is $\mathcal{I}$-reduced iff S_1 and S_2 or T respectively are $\mathcal{I}$-reduced.
- If $S = \mu T.f(T)$, then S is $\mathcal{I}$-reduced iff $f^n(loop)$ is $\mathcal{I}$-reduced for each $n \in \mathbb{N}$.

□

With these technical preliminaries we may now state and prove the upper bound theorem. The proof itself is done by lengthy structural induction on guarded commands and therefore shifted to Appendix C.

Theorem 1 *Let $\mathcal{I}$ be an invariant on X and let S be some $\mathcal{I}$-reduced Y-command with $Y \subseteq X$. Let $S'_{\mathcal{I}}$ result from S as follows:*

- *Each restricted choice $S_1 \boxtimes S_2$ occurring within S will be replaced by $S_1 \square wlp(S_1)(false) \to S_2$.*
- *Then each basic command, i.e. skip, fail, loop and all assignments, will be replaced by their GCSs with respect to $\mathcal{I}$.*

Then $T \sqsubseteq S'_{\mathcal{I}}$ holds for each consistent specialization $T \sqsubseteq S$ with respect to $\mathcal{I}$. □

4.3 The General Form of a GCS

Theorem 1 has a flavour of compositionality, but it does not yet give the GCS. The idea of the main theorem on GCSs is to cut out from the upper bound $S'_{\mathcal{I}}$ those executions that are not allowed to occur in a specialization of S. This is accomplished by adding a precondition $\mathcal{P}$ whose meaning becomes obvious by Proposition 3. This leads to the following theorem.

Theorem 2 *Let $\mathcal{I}$, S and $S'_{\mathcal{I}}$ be as in Theorem 1. Let Z be a disjoint copy of the state space Y. With the formula*

$$\mathcal{P}(S, \mathcal{I}, \vec{x}') \equiv \{\vec{z}/\vec{y}\}.wlp(S''_{\mathcal{I}}; \vec{z} = \vec{x}' \to skip)(wlp(S)^*(\vec{z} = \vec{y})) \quad ,$$

where $S''_{\mathcal{I}}$ results from $S'_{\mathcal{I}}$ by renaming the Y to Z, the GCS $S_{\mathcal{I}}$ is semantically equivalent to

$$@\vec{x}' \bullet \mathcal{P}(S, \mathcal{I}, \vec{x}') \to (S'_{\mathcal{I}}; \vec{y} = \vec{x}' \to skip) \quad .$$

Proof. We take the form claimed in the theorem as a definition and verify the conditions in the definition of the GCS. If φ is an arbitrary Y-formula, we use the definition of dual predicate transformers to validate

$$wlp(S_{\mathcal{I}})^*(\varphi) \quad \Leftrightarrow \quad \exists \vec{x}'.\mathcal{P}(S,\mathcal{I},\vec{x}') \wedge wlp(S'_{\mathcal{I}})^*(\vec{y}=\vec{x}' \wedge \varphi) \quad .$$

If $\mathcal{P}(S,\mathcal{I},\vec{x}')$ holds, then

$$wlp(S'_{\mathcal{I}})^*(\vec{y}=\vec{x}' \wedge \varphi) \quad \Rightarrow \quad wlp(S)^*(\varphi)$$

is *true* for all Y-formulae φ by Proposition 3. But then it follows immediately that $wlp(S_{\mathcal{I}})^*(\varphi) \Rightarrow wlp(S)^*(\varphi)$ holds, hence $S_{\mathcal{I}} \sqsubseteq S$.
Consistency can be verified easily, since $S'_{\mathcal{I}}$ is already consistent with respect to $\mathcal{I}$, namely

$$\begin{array}{lcl}
\mathcal{I} & \Rightarrow & wlp(S'_{\mathcal{I}})(\mathcal{I}) \\
& \Rightarrow & wlp(S'_{\mathcal{I}})(\vec{y}=\vec{x}' \Rightarrow wlp(skip)(\mathcal{I})) \\
& \Leftrightarrow & wlp(S'_{\mathcal{I}})(wlp(\vec{y}=\vec{x}' \rightarrow skip)(\mathcal{I})) \\
& \Leftrightarrow & wlp(S'_{\mathcal{I}};\vec{y}=\vec{x}' \rightarrow skip)(\mathcal{I}) \\
& \Rightarrow & \forall \vec{x}'.\mathcal{P}(S,\mathcal{I},\vec{x}') \Rightarrow wlp(S'_{\mathcal{I}};\vec{y}=\vec{x}' \rightarrow skip)(\mathcal{I}) \\
& \Leftrightarrow & wlp(@\vec{x}' \bullet \mathcal{P}(S,\mathcal{I},\vec{x}') \rightarrow S'_{\mathcal{I}};\vec{y}=\vec{x}' \rightarrow skip)(\mathcal{I}) \\
& \Leftrightarrow & wlp(S_{\mathcal{I}})(\mathcal{I}) \quad .
\end{array}$$

Therefore we have the consistency of $S_{\mathcal{I}}$ with respect to $\mathcal{I}$. Note, that the second implication in the computation above holds due to the monotonicity of $wlp(S'_{\mathcal{I}})$ applied to $\mathcal{I} \Rightarrow (\vec{y}=\vec{x}' \Rightarrow \mathcal{I})$.
Finally, let T be an arbitrary consistent specialization of S. We assume without loss in generality that $wp(T)(true) \Leftrightarrow true$ holds. From Theorem 1 we already get $T \sqsubseteq S'_{\mathcal{I}}$. From this we compute

$$\begin{array}{lcl}
wlp(\underbrace{S'_{\mathcal{I}};\vec{y}=\vec{x}' \rightarrow skip}_{S_{\mathcal{I}}^{\vec{x}'}})(\varphi) & \Leftrightarrow & wlp(S'_{\mathcal{I}})(wlp(\vec{y}=\vec{x}' \rightarrow skip)(\varphi)) \\
& \Rightarrow & wlp(T)(wlp(\vec{y}=\vec{x}' \rightarrow skip)(\varphi)) \\
& \Leftrightarrow & wlp(\underbrace{T;\vec{y}=\vec{x}' \rightarrow skip}_{T^{\vec{x}'}})(\varphi) \quad ,
\end{array}$$

i.e. $T^{\vec{x}'} \sqsubseteq S_{\mathcal{I}}^{\vec{x}'}$. At this point it suffices to show $wp(T^{\vec{x}'})^*(true) \Rightarrow \mathcal{P}(S,\mathcal{I},\vec{x}')$, because

$$\begin{array}{lcl}
wlp(\mathcal{P}(S,\mathcal{I},\vec{x}') \rightarrow S_{\mathcal{I}}^{\vec{x}'})(\varphi) & \Leftrightarrow & \mathcal{P}(S,\mathcal{I},\vec{x}') \Rightarrow wlp(S_{\mathcal{I}}^{\vec{x}'})(\varphi) \\
& \Rightarrow & wp(T^{\vec{x}'})^*(true) \Rightarrow wlp(S_{\mathcal{I}}^{\vec{x}'})(\varphi) \\
& \Rightarrow & wp(T^{\vec{x}'})^*(true) \Rightarrow wlp(T^{\vec{x}'})(\varphi) \\
& \Leftrightarrow & wlp(\underbrace{wp(T^{\vec{x}'})^*(true) \rightarrow T^{\vec{x}'}}_{T^{\vec{x}'}})(\varphi)
\end{array}$$

implies immediately $T^{\vec{x}'} \sqsubseteq \mathcal{P}(S,\mathcal{I},\vec{x}') \rightarrow S_{\mathcal{I}}^{\vec{x}'}$ and we obtain $\forall \vec{x}' \bullet T^{\vec{x}'} \sqsubseteq \forall \vec{x}' \bullet \mathcal{P}(S,\mathcal{I},\vec{x}') \rightarrow S_{\mathcal{I}}^{\vec{x}'}$, consequently. The formula on the left-hand side is equivalent to T, whereas the one on the right-hand side is equivalent to $S_{\mathcal{I}}$.
Assume there is a state $\vec{a}$, in which $\mathcal{P}(S,\mathcal{I},\vec{x}')$ does not hold. From Proposition 3 we get the existence of a state $\vec{b}$ with

$$\models_{\vec{a}} \neg \Big(wlp(S)(\vec{y} \neq \vec{b}) \Rightarrow wlp(S'_{\mathcal{I}}; \vec{y} := \vec{x}' \rightarrow skip)(\vec{y} \neq \vec{b}) \Big) \quad ,$$

which is equivalent to

$$\models_{\vec{a}} wlp(S)(\vec{y} \neq \vec{b}) \wedge \neg wlp(S'_{\mathcal{I}})(\vec{y} = \vec{x}' \Rightarrow \vec{y} \neq \vec{b})$$

and this, finally, to

$$\models_{\vec{a}} wlp(S)(\vec{y} \neq \vec{b}) \wedge wlp(S'_{\mathcal{I}})^*(\vec{y} = \vec{x}' \wedge \vec{y} = \vec{b}) \quad .$$

Hence $\vec{x}' = \vec{b}$ must hold by definition of characterizing state formulae. On the other hand we receive $\models_{\vec{a}} wlp(T)(\vec{y} \neq \vec{b})$ due to $T \sqsubseteq S$ and together with

$$\begin{aligned} wlp(T^{\vec{x}'})(false) \quad &\Leftrightarrow \quad wlp(T)(\vec{y} = \vec{x}' \Rightarrow false) \\ &\Leftrightarrow \quad wlp(T)(\vec{y} \neq \vec{x}') \\ &\Leftrightarrow \quad wlp(T)(\vec{y} \neq \vec{b}) \end{aligned}$$

we conclude $\models_{\vec{a}} wlp(T^{\vec{x}'})(false)$. From the pairing condition $wp(T^{\vec{x}'})(false) \Leftrightarrow wlp(T^{\vec{x}'})(false) \wedge wp(T^{\vec{x}'})(true)$ and

$$wp(T^{\vec{x}'})(true) \;\Leftrightarrow\; wp(T)(\vec{y} = \vec{x}' \Rightarrow true) \;\Leftrightarrow\; wp(T)(true) \;\Leftrightarrow\; true$$

follows $\models_{\vec{a}} wp(T^{\vec{x}'})(false)$, which is equivalent to $\models_{\vec{a}} \neg wp(T^{\vec{x}'})^*(true)$. □

Note that if we consider deterministic branches as a pragmatic approach suggested in [6], then the unbounded choice in Theorem 2 disappears. We omit further details.

The charaterization of GCSs according to Theorem 2 makes it formally possible to reduce consistency enforcement to a simple syntactical replacement (the forming of $S'_{\mathcal{I}}$) and to an investigation of a guard, namely $\mathcal{P}(S,\mathcal{I},\vec{x}')$.

5 Computability and Decidability

We have now reached the stage, where we can say that the GCS approach could have been succesfully developed with respect to arithmetic logic. Thus, we can turn to the original intention of this paper: computability and decidability issues.

Taking the general form of the GCS in Theorem 2 we may now ask, whether we can find an algorithm to compute the GCS. We may further ask, whether the result is effective. In general it will not be possible to compute the GCS, but we will identify subcases, for which effective GCSs can be computed.

5.1 The Computability of GCSs

First consider the computability problem. Taking our Gödel numberings h for terms and formulae and g for commands, we have already exploited their inversibility. From this we obtain the following immediate consequence.

Lemma 5 *For each $n \in \mathbb{N}$ it is decidable, whether n is the Gödel number of a term, a formula or a guarded command.* □

Next we consider the upper bound $S'_{\mathcal{I}}$ that occurs in the GCS. Since this is only a syntactic transformation, we may now conclude that $(S, \mathcal{I}) \mapsto S'_{\mathcal{I}}$ is computable. Hence it is sufficient to investigate the computability for the precondition $\mathcal{P}(S, \mathcal{I}, \vec{x}')$ for arbitrary $\vec{x}'$.

These conditions involve the predicate transformers $wlp(S)$ and $wlp(S'_{\mathcal{I}})$. According to our definition of axiomatic semantics for commands, we know that building these predicate transformers is simple done by syntactic replacement operations. By exploiting our Gödel numbering h again, we conclude that for recursion-free S the mapping

$$(S, \mathcal{I}, \vec{x}') \mapsto \mathcal{P}(S, \mathcal{I}, \vec{x}')$$

– and hence $(S, \mathcal{I}) \mapsto S_{\mathcal{I}}$, too – is computable.
However, if S involves a loop, then $S'_{\mathcal{I}}$ also involves a loop. In order to determine $wlp(S)$ and $wlp(S'_{\mathcal{I}})$ we have to use the limit operator. For a loop $\mu T_j.f(T_j)$ this means to build $wlp(f^i(loop))$ for all $i \in \mathbb{N}$. This is only possible, if there is some $n \in \mathbb{N}$ such that $wlp(f^n(loop)) = wlp(f^m(loop))$ holds for all $m \geq n, m \in \mathbb{N}$. This means that we have a bounded loop (or equivalently a FOR-loop).

Proposition 5 *If recursive guarded commands are restricted to bounded loops, then GCSs are computable, i.e. the function $(S, \mathcal{I}) \mapsto S_{\mathcal{I}}$ is computable. In general, however, the GCS cannot be computed.* □

5.2 Effective GCSs

Even, if the GCS $S_{\mathcal{I}}$ can be computed from a given command S and the invariant $\mathcal{I}$, the result still contains the preconditions $\mathcal{P}(S, \mathcal{I}, \vec{x}')$. If such a precondition is undecidable, then the GCSs will not be effective. We will demonstrate how effective GCSs can be computed.

Therefore, we consider the proof of the upper bound theorem (see Appendix C) again. The next result shows that we have already proven more than we needed.

Lemma 6 *Let T be a program specification on Y and $\mathcal{I}$ a static constraint on X with $Y \subseteq X$.*

1. *If $T = P \to S$, then $T_{\mathcal{I}} = P \to S_{\mathcal{I}}$.*
2. *If $T = S_1 \square S_2$, then $T_{\mathcal{I}} = (S_1)_{\mathcal{I}} \square (S_2)_{\mathcal{I}}$.*
3. *If $T = @y \bullet S$, then $T_{\mathcal{I}} = @y \bullet S_{\mathcal{I}}$.*

Proof. The Propositions 7, 8 and 9 show the specialization in one direction. For the reverse specialization, one shows straightforwardly that $P \to S_{\mathcal{I}}$, $(S_1)_{\mathcal{I}} \square (S_2)_{\mathcal{I}}$ and $@y \bullet S_{\mathcal{I}}$ are $\mathcal{I}$-consistent specializations of $P \to S$, $S_1 \square S_2$ and $@y \bullet S$, respectively. □

Note, that Lemma 6 does not hold for the case of sequences, even if they are δ-$\mathcal{I}$-reduced. Although Proposition 11 gives us of course specialization in one direction, the reverse specialization does not hold in general. The reason why $(S_1)_{\mathcal{I}}; (S_2)_{\mathcal{I}}$ is not a specialization of $S_1; S_2$ is that $wlp(S_2)(\varphi)$ is not necessarily a state formula of the underlying S_1 state space.

The next lemma will give us a computation of effective GCSs for program specifications S that only use basic commands, choices, guards and sequences. We dispense with the case of restricted choices.

Lemma 7 *Let S be a program specification on X built of basic commands, choices, guards with decidable preconditions and sequences. If φ is a decidable state formula on X, then $wlp(S)(\varphi)$ and $wlp(S)^*(\varphi)$ are decidable as well.*

Proof. The proof is a straightforward structural induction that makes use of the closure properties for decidable arithmetical predicates. □

It it well-known that every first-order predicate formula φ is equivalent to a formula $Q_1 x_1 \ldots Q_k x_k . \psi$ where $Q_i \in \{\forall, \exists\}$ for $i = 1, \ldots, k$ and ψ is quantifier-free. This result carries immediately over to guarded commands with respect to the @-operator.

Lemma 8 *Each guarded command S, whose occurences of loops are all bounded, can be written in the form $@x_1 \bullet \ldots @x_n \bullet S'$ such that S' does not contain an unbounded choice operator @.*

Proof. The only interesting case is the one for bounded loops. Applying the predicate transformer *wlp* here results in a finite conjunction, whereas *wp* gives a finite disjunction. □

Let us all bring together and consider a program specification S for which all occurences of loops are bounded and all preconditions are decidable. In a first step, we replace all occurences of the restricted choice operator $\boxtimes$ in the usual way. Then we apply Lemma 8 that provides us with a specification $T = @y_1 \bullet \ldots @y_n \bullet R$ that is semantically equivalent to S. Lemma 6 tells us then not to worry about the occurences of unbounded-choice operators, i.e., $T_{\mathcal{I}} = @y_1 \bullet \ldots @y_n \bullet R_{\mathcal{I}}$. We apply the main theorem (Theorem 2) to compute $R_{\mathcal{I}}$ and conclude by Lemma 7 that all preconditions of the form $\mathcal{P}(S', \mathcal{I}, \vec{x}')$ are decidable. Finally, we obtain the following result.

Proposition 6 *Let S be a program specification such that every loop is bounded and all preconditions are decidable. Let $\mathcal{I}$ be a decidable static constraint. Then we can compute the GCS $S_{\mathcal{I}}$ in the form $S_{\mathcal{I}} = @y_1 \bullet \ldots @y_n \bullet T_{\mathcal{I}}$, where $T_{\mathcal{I}}$ has the form of Theorem 2 with all preconditions $\mathcal{P}(T', \mathcal{I}, \vec{x}')$ being decidable.* □

6 Conclusion

In this article we considered the GCS approach to consistency enforcement presented in [6]. We could show that the underlying theory of predicate transformers could be carried over from an infinitary logic to first-order arithmetic logic. We were even able to do this for recursive program specifications by exploiting Gödel numberings for terms, formulae and guarded commands. However, the used recursive program specifications are slightly restricted with respect to the more general theory in [4].

Then we could show that the existence and uniqueness of GCSs, the commutativity result from [8] and the fundamental compositionality result carry over to the new logic. This allows to study computability and decidability issues. We could show that the GCS is computable for program specifications where all loops are bounded. Moreover, effective GCSs can be computed when preconditions within guards and the given static constraint are decidable.

There are at least three more problems we would like to approach next. Firstly, we would like to study the Goldfarb classification [2] and its impact to GCS construction. More precisely, we look for a characterization of those static invariants $\mathcal{I}$ for which $\mathcal{I}$-reducedness is decidable. Secondly, we would like to look at weakened approaches to consistency enforcement, e.g. the one presented in [5] and to discuss computability and decidability for this approach as well. Thirdly and finally, we would like to address the problems of GCSs – and weakened approaches – with respect to basic commands. In particular, it would be nice to see how GCSs for various classes of relational constraints would look like.

A Appendix A: Proof of the Normal Form for Specialization

Proposition 3. *Let S and T be commands on the state spaces X and Y, respectively, with $X \subseteq Y$. Then $wlp(S)(\varphi) \Rightarrow wlp(T)(\varphi)$ holds for all X-formulae iff*

$$\{\vec{z}/\vec{x}\}.wlp(T')(wlp(S)^*(\vec{x} = \vec{z}))$$

holds, where $\vec{z}$ is a disjoint copy of $\vec{x}$ and T' results from T by renaming each x_i into z_i.

Proof. The normal form representation from Lemma 1 gives for $wlp(T')$ the equivalence from $wlp(T')(wlp(S)^*(\vec{x} = \vec{z}))$ to

$$\forall \vec{z}'.wlp(T')^*(\vec{z} = \vec{z}') \Rightarrow \{\vec{z}/\vec{z}'\}.wlp(S)^*(\vec{x} = \vec{z}).$$

Now, S is defined on X which results in

$$\{\vec{z}/\vec{z}'\}.wlp(S)^*(\vec{x} = \vec{z}) \quad \Leftrightarrow \quad wlp(S)^*(\vec{x} = \vec{z}') \quad .$$

Hence, it is sufficient to show the equivalence between

1. $wlp(S)(\varphi) \quad \Rightarrow \quad wlp(T)(\varphi)$ for all X-formulae φ and
2. $\{\vec{z}/\vec{x}\}.(\forall \vec{z}'.wlp(T')^*(\vec{z} = \vec{z}') \Rightarrow wlp(S)^*(\vec{x} = \vec{z}'))$.

Let us assume that (1) holds. By renaming, $wlp(S')(\varphi) \Rightarrow wlp(T')(\varphi)$ holds for all Z-formulae φ. In particular, if $\varphi \equiv \vec{z} = \vec{a}$ for some state $\vec{a}$, then $wlp(S')(\vec{z} = \vec{a}) \Leftrightarrow \{\vec{x}/\vec{z}\}.wlp(S)^*(\vec{x} = \vec{a})$. But then,

$$\forall \vec{z}'.(wlp(T')^*(\vec{z} = \vec{z}') \Rightarrow \{\vec{x}/\vec{z}\}.wlp(S)^*(\vec{x} = \vec{z}'))$$

must be valid and this implies (2).
Suppose that (2) holds. Again, Lemma 1 can be employed to show the equivalence of $wlp(T)^*(\varphi)$ with arbitrary X-formula φ to

$$\exists \vec{z}'.(\{\vec{z}/\vec{x}\}.wlp(T')^*(\vec{z} = \vec{z}') \wedge \varphi(\vec{z}')) \quad .$$

With $\forall \vec{z}'.(wlp(T')^*(\vec{z} = \vec{z}') \Rightarrow \{\vec{x}/\vec{z}'\}.wlp(S)^*(\vec{x} = \vec{z}'))$ follows immediately

$$\{\vec{z}/\vec{x}\}.(\exists \vec{z}'.(wlp(S)^*(\vec{x} = \vec{z}') \wedge \varphi(\vec{z}'))) \quad ,$$

which is equivalent to $wlp(S)^*(\varphi)$ by Lemma 1. This gives the proof. □

B Appendix B: Existence, Normal Form Representation and Commutativity of GCSs

In the appendix we give a detailed proof of Proposition 4.

Proposition 4. *The GCS $S_{\mathcal{I}}$ of S with respect to $\mathcal{I}$ always exists and is unique up to semantic equivalence. We can always write*

$$S_{\mathcal{I}} = (\mathcal{I} \rightarrow (S; @\vec{z}' \bullet \vec{z} := \vec{z}'; \mathcal{I} \rightarrow skip)) \boxtimes (\neg\mathcal{I} \rightarrow (S; @\vec{z}' \bullet \vec{z} := \vec{z}')) \ ,$$

where $\vec{z}$ refers to the free variables in $\mathcal{I}$ not occurring in S.

Furthermore, for two invariants $\mathcal{I}$ and $\mathcal{J}$ we always obtain that $\mathcal{I} \wedge \mathcal{J} \rightarrow S_{\mathcal{I}\wedge\mathcal{J}}$ and $\mathcal{I} \wedge \mathcal{J} \rightarrow (S_{\mathcal{I}})_{\mathcal{J}}$ are semantically equivalent.

Proof. First we show the existence and uniqueness up to semantic equivalence of GCS. We set

$$\mathcal{T} = \{T \mid T \sqsubseteq S \text{ and } T \text{ is consistent with respect to } \mathcal{I}\} \quad .$$

If the least upper bound $S_{\mathcal{I}}$ of $\mathcal{T}$ with respect to the specialization $\sqsubseteq$ exists, then this must be the GCS. Therefore, we have the uniqueness up to semantic equivalence. We now verify the conditions from Definition 4 for the program specification $S_{\mathcal{I}}$ above. Let φ be an arbitrary state formula on Y. Then we receive

$$\begin{aligned}
wlp(S_{\mathcal{I}})^*(\varphi) \quad &\Leftrightarrow \quad (\mathcal{I} \wedge wlp(S)^*(\exists \vec{z}'.\{\vec{z}/\vec{z}'\}.(\mathcal{I} \wedge \varphi))) \vee \\
&\qquad (\neg\mathcal{I} \wedge wlp(S)^*(\exists \vec{z}'.\{\vec{z}/\vec{z}'\}.\varphi)) \\
&\Leftrightarrow \quad (\mathcal{I} \wedge wlp(S)^*((\exists \vec{z}'.\{\vec{z}/\vec{z}'\}.\mathcal{I}) \wedge \varphi)) \vee (\neg\mathcal{I} \wedge wlp(S)^*(\varphi)) \\
&\Rightarrow \quad (\mathcal{I} \wedge wlp(S)^*(\varphi)) \vee (\neg\mathcal{I} \wedge wlp(S)^*(\varphi)) \\
&\Leftrightarrow \quad wlp(S)^*(\varphi) \quad .
\end{aligned}$$

Doing this we have made use of the dual predicate transformers' monotonicity property and the fact that variables z_i do not occur within φ. Then the asserted specialization $S_{\mathcal{I}} \sqsubseteq S$ follows from the same computation for wp instead of wlp.
Next we consider

$$\begin{aligned} wlp(S_{\mathcal{I}})(\mathcal{I}) \quad &\Leftrightarrow \quad (\mathcal{I} \Rightarrow wlp(S)(\forall \vec{z}'.\{\vec{z}/\vec{z}'\}.(\mathcal{I} \Rightarrow \mathcal{I}))) \wedge \\ &\qquad\quad (\neg\mathcal{I} \Rightarrow wlp(S)(\forall \vec{z}'.\{\vec{z}/\vec{z}'\}.\mathcal{I})) \\ &\Leftrightarrow \quad \neg\mathcal{I} \Rightarrow wlp(S)(\forall \vec{z}'.\{\vec{z}/\vec{z}'\}.\mathcal{I}) \\ &\Leftrightarrow \quad \mathcal{I} \vee \neg wlp(S)(\forall \vec{z}'.\{\vec{z}/\vec{z}'\}.\mathcal{I}) \quad . \end{aligned}$$

and obtain $\mathcal{I} \Rightarrow wlp(S_{\mathcal{I}})(\mathcal{I})$ which means that the above $S_{\mathcal{I}}$ is indeed consistent with respect to $\mathcal{I}$.
Let $\vec{x} = \vec{y}$ be a characterizing state formula and $T \sqsubseteq S$ an arbitrary, but $\mathcal{I}$-consistent specialization of S. Then we ditinguish two cases.
Case 1. We assume $\vec{x} = \vec{y} \Rightarrow \neg\mathcal{I}$ and therefore we conclude $wlp(T)^*(\vec{x} = \vec{y}) \Rightarrow wlp(T)^*(\neg\mathcal{I}) \Rightarrow \neg\mathcal{I}$ using the monotonicity of $wlp(S)^*$ and consistency of T. Moreover, it follows

$$\begin{aligned} wlp(T)^*(\vec{x} = \vec{y}) \quad &\Rightarrow \quad \neg\mathcal{I} \wedge wlp(S)^*(\vec{x} = \vec{y}) \\ &\Rightarrow \quad \neg\mathcal{I} \wedge wlp(S)^*(\exists \vec{z}'.\{\vec{z}/\vec{z}'\}.\vec{x} = \vec{y}) \\ &\Rightarrow \quad wlp(S_{\mathcal{I}})^*(\vec{x} = \vec{y}) \quad . \end{aligned}$$

For the first implication we simply use the specialization $T \sqsubseteq S$, for the second we refer to the monotonicity applied to $\vec{x} = \vec{y} \Rightarrow \exists \vec{z}'.\{\vec{z}/\vec{z}'\}.\vec{x} = \vec{y}$ and the last one follows from the first line of the computation of $wlp(S_{\mathcal{I}})^*$
Case 2. Starting from $\vec{x} = \vec{y} \Rightarrow \mathcal{I}$ gives $wlp(T)^*(\vec{x} = \vec{y}) \Leftrightarrow wlp(T)^*(\mathcal{I} \wedge \vec{x} = \vec{y})$, subsequentely. We compute the following using $T \sqsubseteq S$ and the monotonicity of $wlp(S)^*$

$$\begin{aligned} wlp(T)^*(\vec{x} = \vec{y}) \quad &\Rightarrow \quad wlp(S)^*(\exists \vec{z}'.\{\vec{z}/\vec{z}'\}.(\mathcal{I} \wedge \vec{x} = \vec{y})) \wedge \\ &\qquad\quad wlp(S)^*(\exists \vec{z}'.\{\vec{z}/\vec{z}'\}.\vec{x} = \vec{y}) \\ &\Rightarrow \quad (\mathcal{I} \wedge wlp(S)^*(\exists \vec{z}'.\{\vec{z}/\vec{z}'\}.(\mathcal{I} \wedge \vec{x} = \vec{y}))) \vee \\ &\qquad\quad (\neg\mathcal{I} \wedge wlp(S)^*(\exists \vec{z}'.\{\vec{z}/\vec{z}'\}.\vec{x} = \vec{y})) \\ &\Leftrightarrow \quad wlp(S_{\mathcal{I}})^*(\vec{x} = \vec{y}) \quad . \end{aligned}$$

This first step has brought us to $wlp(T)^*(\vec{x} = \vec{y}) \Rightarrow wlp(S_{\mathcal{I}})^*(\vec{x} = \vec{y})$, i.e. $wlp(S_{\mathcal{I}})(\vec{x} \neq \vec{y}) \Rightarrow wlp(T)(\vec{x} \neq \vec{y})$. For arbitrary state formula φ we have $\varphi(\vec{x}) \Leftrightarrow \forall \vec{y}.\neg\varphi(\vec{y}) \Rightarrow \vec{x} \neq \vec{y}$ and therefore

$$\begin{aligned} wlp(S_{\mathcal{I}})(\varphi(\vec{x})) \quad &\Leftrightarrow \quad \forall \vec{y}.\neg\varphi(\vec{y}) \Rightarrow wlp(S_{\mathcal{I}})(\vec{x} \neq \vec{y}) \\ &\Rightarrow \quad \forall \vec{y}.\neg\varphi(\vec{y}) \Rightarrow wlp(T)(\vec{x} \neq \vec{y}) \\ &\Leftrightarrow \quad wlp(T)(\varphi(\vec{x})) \quad , \end{aligned}$$

using the universal conjunctivity property of wlp. Thus, we obtain $wlp(T)^*(\varphi) \Rightarrow wlp(S_{\mathcal{I}})^*(\varphi)$ for all φ. On top of that $wp(T)^*(false) \Rightarrow wp(S)^*(false) \Rightarrow$

$wp(S_{\mathcal{I}})^*(false)$ holds as well, due to the specialization $T \sqsubseteq S$ and the first line of the computation of $wlp(S_{\mathcal{I}})^*$ above. Indeed, we have proved that T is a specialization of $S_{\mathcal{I}}$.
Let us now consider the asserted commutativity result. Since $(S_{\mathcal{I}_1})_{\mathcal{I}_2}$ is $\mathcal{I}_2$-consistent by definition we have

$$\mathcal{I}_2 \quad \Rightarrow \quad wlp((S_{\mathcal{I}_1})_{\mathcal{I}_2})(\mathcal{I}_2) \quad .$$

On the other side we can use the definition of GCS and consistency as well as $(S_{\mathcal{I}_1})_{\mathcal{I}_2} \sqsubseteq S_{\mathcal{I}_1}$ in order to receive

$$\mathcal{I}_1 \quad \Rightarrow \quad wlp(S_{\mathcal{I}_1})(\mathcal{I}_1) \quad \Rightarrow \quad wlp((S_{\mathcal{I}_1})_{\mathcal{I}_2})(\mathcal{I}_1) \quad .$$

In summary, this results in

$$\mathcal{I}_1 \wedge \mathcal{I}_2 \;\Rightarrow\; wlp((S_{\mathcal{I}_1})_{\mathcal{I}_2})(\mathcal{I}_1) \wedge wlp((S_{\mathcal{I}_1})_{\mathcal{I}_2})(\mathcal{I}_2) \;\Leftrightarrow\; wlp((S_{\mathcal{I}_1})_{\mathcal{I}_2})(\mathcal{I}_1 \wedge \mathcal{I}_2) \quad ,$$

so we have proved the consistency of $(S_{\mathcal{I}_1})_{\mathcal{I}_2}$ with respect to $\mathcal{I}_1 \wedge \mathcal{I}_2$. From $S_{\mathcal{I}_1} \sqsubseteq S$ and $(S_{\mathcal{I}_1})_{\mathcal{I}_2} \sqsubseteq S_{\mathcal{I}_1}$ we derive

$$wlp(S)(\varphi) \quad \Rightarrow \quad wlp(S_{\mathcal{I}_1})(\varphi) \quad \Rightarrow \quad wlp((S_{\mathcal{I}_1})_{\mathcal{I}_2})(\varphi),$$

i.e. the specialization $(S_{\mathcal{I}_1})_{\mathcal{I}_2} \sqsubseteq S$. Consequentely, definition 4 yields $(S_{\mathcal{I}_1})_{\mathcal{I}_2} \sqsubseteq S_{\mathcal{I}_1 \wedge \mathcal{I}_2}$ and we obtain

$$\begin{aligned} wlp(\mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow S_{\mathcal{I}_1 \wedge \mathcal{I}_2})(\varphi) \quad &\Leftrightarrow \quad \mathcal{I}_1 \wedge \mathcal{I}_2 \Rightarrow wlp(S_{\mathcal{I}_1 \wedge \mathcal{I}_2})(\varphi) \\ &\Rightarrow \quad \mathcal{I}_1 \wedge \mathcal{I}_2 \Rightarrow wlp((S_{\mathcal{I}_1})_{\mathcal{I}_2})(\varphi) \\ &\Leftrightarrow \quad wlp(\mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow (S_{\mathcal{I}_1})_{\mathcal{I}_2})(\varphi) \end{aligned}$$

for arbitrary φ which means $\mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow (S_{\mathcal{I}_1})_{\mathcal{I}_2} \;\sqsubseteq\; \mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow S_{\mathcal{I}_1 \wedge \mathcal{I}_2}$. Thus, it remains to show the reverse specialization.
From $S_{\mathcal{I}_1 \wedge \mathcal{I}_2} \sqsubseteq S$ follows

$$\mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow S_{\mathcal{I}_1 \wedge \mathcal{I}_2} \quad \sqsubseteq \quad S \quad . \tag{1}$$

In addition, $S_{\mathcal{I}_1 \wedge \mathcal{I}_2}$ is consistent with respect to $\mathcal{I}_1 \wedge \mathcal{I}_2$ of definition, so we have not only $\mathcal{I}_1 \wedge \mathcal{I}_2 \Rightarrow wlp(S_{\mathcal{I}_1 \wedge \mathcal{I}_2})(\mathcal{I}_1)$ but also $\mathcal{I}_1 \wedge \mathcal{I}_2 \Rightarrow wlp(S_{\mathcal{I}_1 \wedge \mathcal{I}_2})(\mathcal{I}_2)$. Next we consider

$$\mathcal{I}_1 \Rightarrow wlp(\mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow S_{\mathcal{I}_1 \wedge \mathcal{I}_2})(\mathcal{I}_1) \quad \Leftrightarrow \quad \mathcal{I}_1 \wedge \mathcal{I}_2 \Rightarrow wlp(S_{\mathcal{I}_1 \wedge \mathcal{I}_2})(\mathcal{I}_1) \tag{2}$$

and

$$\mathcal{I}_2 \Rightarrow wlp(\mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow S_{\mathcal{I}_1 \wedge \mathcal{I}_2})(\mathcal{I}_2) \quad \Leftrightarrow \quad \mathcal{I}_1 \wedge \mathcal{I}_2 \Rightarrow wlp(S_{\mathcal{I}_1 \wedge \mathcal{I}_2})(\mathcal{I}_2) \quad . \tag{3}$$

From equation (2) we obtain the consistency of $\mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow S_{\mathcal{I}_1 \wedge \mathcal{I}_2}$ with respect to $\mathcal{I}_1$ and using equation (1) yields

$$\mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow S_{\mathcal{I}_1 \wedge \mathcal{I}_2} \quad \sqsubseteq \quad S_{\mathcal{I}_1} \quad . \tag{4}$$

From equation (3) follows the consistency of $\mathcal{I}_1 \wedge \mathcal{I}_2 \to S_{\mathcal{I}_1 \wedge \mathcal{I}_2}$ with respect to $\mathcal{I}_2$ and using equation (4) we conclude

$$\mathcal{I}_1 \wedge \mathcal{I}_2 \to S_{\mathcal{I}_1 \wedge \mathcal{I}_2} \quad \sqsubseteq \quad (S_{\mathcal{I}_1})_{\mathcal{I}_2} \quad . \tag{5}$$

Finally, we compute

$$\begin{aligned} w(l)p(\mathcal{I}_1 \wedge \mathcal{I}_2 \to (S_{\mathcal{I}_1})_{\mathcal{I}_2})(\varphi) \quad &\Leftrightarrow \quad \mathcal{I}_1 \wedge \mathcal{I}_2 \Rightarrow w(l)p((S_{\mathcal{I}_1})_{\mathcal{I}_2})(\varphi) \\ &\Rightarrow \quad \mathcal{I}_1 \wedge \mathcal{I}_2 \Rightarrow w(l)p(\mathcal{I}_1 \wedge \mathcal{I}_2 \to S_{\mathcal{I}_1 \wedge \mathcal{I}_2})(\varphi) \\ &\Leftrightarrow \quad \mathcal{I}_1 \wedge \mathcal{I}_2 \Rightarrow (\mathcal{I}_1 \wedge \mathcal{I}_2 \Rightarrow w(l)p(S_{\mathcal{I}_1 \wedge \mathcal{I}_2})(\varphi)) \\ &\Leftrightarrow \quad \mathcal{I}_1 \wedge \mathcal{I}_2 \Rightarrow w(l)p(S_{\mathcal{I}_1 \wedge \mathcal{I}_2})(\varphi) \\ &\Leftrightarrow \quad w(l)p(\mathcal{I}_1 \wedge \mathcal{I}_2 \to S_{\mathcal{I}_1 \wedge \mathcal{I}_2})(\varphi) \end{aligned}$$

the specialization $\mathcal{I}_1 \wedge \mathcal{I}_2 \to S_{\mathcal{I}_1 \wedge \mathcal{I}_2} \sqsubseteq \mathcal{I}_1 \wedge \mathcal{I}_2 \to (S_{\mathcal{I}_1})_{\mathcal{I}_2}$, where we just make use of equation (5) in the appearing implication. This completes the proof. □

C Appendix C: Proof of the Upper Bound Theorem

Recall the strategy, to obtain a new specification $S'_{\mathcal{I}}$ from a given complex program specification S and static invariant $\mathcal{I}$ by replacing all basic commands, i.e. *skip*, *fail*, *loop* and in particular assignments, within S by their respective GCSs. The upper bound theorem 1 proposes that this yields an upper bound for $S_{\mathcal{I}}$ with respect to the specialization order $\sqsubseteq$, i.e., $S_{\mathcal{I}} \sqsubseteq S'_{\mathcal{I}}$.

The result is only provable if we assume that S is in $\mathcal{I}$-reduced form. We use structural induction on guarded commands and start with $\to, \square, @$ and $\boxtimes$. We will deal with the more difficult cases of sequences and recursion in subsections.

Proposition 7 *Let $S' = P \to S$ be a specification on Y and $\mathcal{I}$ a static constraint on X with $Y \subseteq X$. If $T \sqsubseteq S'$ is $\mathcal{I}$-consistent, then $T \sqsubseteq P \to S_{\mathcal{I}}$.*

Proof. First $w(l)p(S)(\varphi) \Rightarrow (P \Rightarrow w(l)p(S)(\varphi))$ establishes $S' \sqsubseteq S$, hence $T \sqsubseteq S$ by assumption and transitivity of $\sqsubseteq$. Moreover, the $\mathcal{I}$-consistency of T gives us even $T \sqsubseteq S_{\mathcal{I}}$. From

$$wp(S')(false) \quad \Leftrightarrow \quad P \Rightarrow wp(S)(false) \quad \Leftrightarrow \quad \neg P \vee wp(S)(false)$$

we receive $\neg P \Rightarrow wp(S')(false)$. As the specialization $T \sqsubseteq S'$ means in particular $wp(S')(false) \Rightarrow wp(T)(false)$, we conclude $\neg P \Rightarrow wp(T)(false)$ or equivalently $wp(T)^*(true) \Rightarrow P$. But then

$$\begin{aligned} w(l)p(\mathcal{P} \to S_{\mathcal{I}})(\varphi) \quad &\Leftrightarrow \quad P \Rightarrow w(l)p(S_{\mathcal{I}})(\varphi) \\ &\Rightarrow \quad P \Rightarrow w(l)p(T)(\varphi) \\ &\Rightarrow \quad wp(T)^*(true) \Rightarrow w(l)p(T)(\varphi) \\ &\Leftrightarrow \quad w(l)p(wp(T)^*(true) \to T)(\varphi) \\ &\Leftrightarrow \quad w(l)p(T)(\varphi) \quad , \end{aligned}$$

holds and therefore the desired specialization $T \sqsubseteq P \to S_{\mathcal{I}}$. □

Proposition 8 *Let $S = S_1 \square S_2$ be a program specification on Y and $\mathcal{I}$ a static invariant on X with $Y \subseteq X$. If $T \sqsubseteq S$ is $\mathcal{I}$-consistent, then $T \sqsubseteq (S_1)_{\mathcal{I}} \square (S_2)_{\mathcal{I}}$.*

Proof. We start showing the semantic equivalence of T to $T' \square \mathcal{Q} \to loop$ with $wp(T')(true) \Leftrightarrow true$, $wlp(T')(\varphi) \Leftrightarrow wlp(T)(\varphi)$ for arbitrary φ and $\mathcal{Q} \Leftrightarrow wp(T)^*(false)$. Namely,

$$\begin{array}{rcll} wlp(T' \square \mathcal{Q} \to loop)(\varphi) & \Leftrightarrow & wlp(T')(\varphi) \wedge (\mathcal{Q} \Rightarrow wlp(loop)(\varphi)) & \\ & \Leftrightarrow & wlp(T)(\varphi) \wedge true & \\ & \Leftrightarrow & wlp(T)(\varphi) & \text{and} \\ wp(T' \square \mathcal{Q} \to loop)(\varphi) & \Leftrightarrow & wp(T')(\varphi) \wedge (\mathcal{Q} \Rightarrow wp(loop)(\varphi)) & \\ & \Leftrightarrow & wlp(T')(\varphi) \wedge wp(T')(true) \wedge \neg \mathcal{Q} & \\ & \Leftrightarrow & wlp(T)(\varphi) \wedge \neg wp(T)^*(false) & \\ & \Leftrightarrow & wlp(T)(\varphi) \wedge wp(T)(true) & \\ & \Leftrightarrow & wp(T)(\varphi). & \end{array}$$

From

$$w(l)p(S)(\varphi) \Rightarrow w(l)p(T)(\varphi) \quad \Leftrightarrow \quad w(l)p(T')(\varphi) \wedge w(l)p(\mathcal{Q} \to loop)(\varphi)$$

we obtain $\mathcal{Q} \to loop \sqsubseteq S$ and therefore also

$$\mathcal{Q} \to loop \;=\; (\mathcal{Q}_1 \to loop) \square (\mathcal{Q}_2 \to loop),$$

with $\mathcal{Q}_i \to loop \sqsubseteq S_i$ for $i = 1, 2$. We show $T' \sqsubseteq (S_1)_{\mathcal{I}} \square (S_2)_{\mathcal{I}}$ since this implies

$$T \;\sqsubseteq\; \underbrace{(S_1)_{\mathcal{I}} \square (\mathcal{Q}_1 \to loop)}_{(S_1)'_{\mathcal{I}}} \square \underbrace{(S_2)_{\mathcal{I}} \square (\mathcal{Q}_2 \to loop)}_{(S_2)'_{\mathcal{I}}}$$

with $(S_i)'_{\mathcal{I}} \sqsubseteq (S_i)_{\mathcal{I}}$ for $i = 1, 2$. Namely, $\mathcal{Q}_i \to loop \sqsubseteq S_i$, $(S_i)_{\mathcal{I}} \sqsubseteq S_i$ implies $(S_i)'_{\mathcal{I}} \sqsubseteq S_i$ and from the $\mathcal{I}$-consistency of $(S_i)'_{\mathcal{I}}$ follows $(S_i)'_{\mathcal{I}} \sqsubseteq (S_i)_{\mathcal{I}}$.

Without loss in generality we assume that $wp(T)(true) \Leftrightarrow true$ holds. For each state $\vec{a}$ on Y we define $T^{\vec{a}} = T; (\vec{y} = \vec{a} \to skip)$. Then $T^{\vec{a}}$ is a deterministic specialization of T as

$$\begin{array}{rcll} wlp(T^{\vec{a}})^*(\vec{y} = \vec{b}) & \Leftrightarrow & wlp(T)^*(\vec{y} = \vec{a} \wedge \vec{y} = \vec{b}) & \\ & \Leftrightarrow & \left\{ \begin{array}{ll} wlp(T)^*(\vec{y} = \vec{a}) & \text{for } \vec{b} = \vec{a} \\ false & \text{otherwise} \end{array} \right. & \\ & \Rightarrow & wlp(T)^*(\vec{y} = \vec{b}) & \text{and} \\ wp(T^{\vec{a}})^*(\vec{y} = \vec{b}) & \Leftrightarrow & wp(T)^*(\vec{y} = \vec{a} \wedge \vec{y} = \vec{b}) & \\ & \Leftrightarrow & \left\{ \begin{array}{ll} wp(T)^*(\vec{y} = \vec{a}) & \text{for } \vec{b} = \vec{a} \\ wp(T)^*(false) & \text{otherwise} \end{array} \right. & \\ & \Rightarrow & wp(T)^*(\vec{y} = \vec{b}) \quad . & \end{array}$$

The last implication in the second case follows from the monotonicity of $wlp(T)^*$ applied to $false \Rightarrow \vec{y} = \vec{b}$. Besides, we obtain $w(l)p(T^{\vec{a}})^*(R) \Rightarrow w(l)p(T)^*(R)$ for arbitrary φ. From $\varphi(\vec{y}) \Leftrightarrow \forall \vec{z}.\neg\varphi(\vec{z}) \Rightarrow \vec{y} \neq \vec{z}$ we derive

$$\begin{aligned} wlp(T)(\varphi(\vec{y})) \quad &\Leftrightarrow \quad wlp(T)(\forall \vec{z}.\neg\varphi(\vec{z}) \Rightarrow \vec{y} \neq \vec{z}) \\ &\Leftrightarrow \quad \forall \vec{z}.\neg\varphi(\vec{z}) \Rightarrow wlp(T)(\vec{y} \neq \vec{z}) \\ &\Rightarrow \quad \forall \vec{z}.\neg\varphi(\vec{z}) \Rightarrow wlp(T^{\vec{a}})(\vec{y} \neq \vec{z}) \\ &\Leftrightarrow \quad wlp(T^{\vec{a}})(\forall \vec{z}.\neg\varphi(\vec{z}) \Rightarrow \vec{y} \neq \vec{z}) \\ &\Leftrightarrow \quad wlp(T^{\vec{a}})(\varphi(\vec{y})) \quad , \end{aligned}$$

i.e., the specialization $T^{\vec{a}} \sqsubseteq T$, as the wp-part can be obtain similarily. Here, in case of an empty index set we use $wp(T^{\vec{a}})(true) \Leftrightarrow wp(T)(true)$. The proof that $T^{\vec{a}}$ is deterministic uses

$$wlp(T^{\vec{a}})^*(\vec{y} = \vec{b}) \quad \Leftrightarrow \quad wlp(T)^*(\vec{y} = \vec{b} \wedge \vec{y} = \vec{a})$$

and the distinction into two cases. If $\vec{b} \neq \vec{a}$ holds, then

$$wlp(T)^*(\vec{y} = \vec{b} \wedge \vec{y} = \vec{a}) \Leftrightarrow wlp(T)^*(false) \Leftrightarrow false \Rightarrow wp(T^{\vec{a}})(\vec{y} = \vec{b})$$

and if $\vec{b} = \vec{a}$ is valid, then

$$wp(T^{\vec{a}})(\vec{y} = \vec{a}) \Leftrightarrow wp(T)(\vec{y} = \vec{a} \Rightarrow \vec{y} = \vec{a}) \Leftrightarrow true$$

implies $wp(T^{\vec{a}})(\vec{y} = \vec{b})$. Together $wlp(T^{\vec{a}})^*(\varphi) \Rightarrow wp(T^{\vec{a}})(\varphi)$ for arbitrary φ means that $T^{\vec{a}}$ is deterministic. Using wlp's monotonicity, we conclude $\mathcal{I} \Rightarrow wlp(T)(\mathcal{I}) \Rightarrow wlp(T)(\vec{y} = \vec{a} \Rightarrow \mathcal{I}) \Rightarrow wlp(T^{\vec{a}})(\mathcal{I})$ and therefore that $T^{\vec{a}}$ is also $\mathcal{I}$-consistent. As we have just proven that $T^{\vec{a}}$ is deterministic, it is also semantically equivalent to $T_1^{\vec{a}} \Box T_2^{\vec{a}}$ with $T_i^{\vec{a}} \sqsubseteq S_i$ for $i = 1, 2$. More precisely, we have $T_i^{\vec{a}} = P_i^{\vec{a}} \rightarrow T^{\vec{a}}$ with

$$P_i^{\vec{a}} \Leftrightarrow \{\vec{z}/\vec{y}\}.wlp(\{\vec{y}/\vec{z}\}.T^{\vec{a}})(\vec{z} = \vec{a} \Rightarrow wlp(S_i)^*(\vec{z} = \vec{y})).$$

Using Proposition 3 we have

$$P_1^{\vec{a}} \vee P_2^{\vec{a}} \Leftrightarrow \{\vec{z}/\vec{y}\}.wlp(\{\vec{y}/\vec{z}\}.T^{\vec{a}})(\vec{y} = \vec{a} \Rightarrow wlp(S)^*(\vec{z} = \vec{y})) \Leftrightarrow true \quad ,$$

where $T^{\vec{a}} \sqsubseteq S$ is applied. Moreover, $T^{\vec{a}} = (P_1^{\vec{a}} \vee P_2^{\vec{a}}) \rightarrow T^{\vec{a}} = P_1^{\vec{a}} \rightarrow T^{\vec{a}} \Box P_2^{\vec{a}} \rightarrow T^{\vec{a}}$ holds. Since $T_i^{\vec{a}}$ is $\mathcal{I}$-consistent for $i = 1, 2$, the GCS definition gives us $T_i^{\vec{a}} \sqsubseteq (S_i)_{\mathcal{I}}$ and therefore $T^{\vec{a}} \sqsubseteq (S_1)_{\mathcal{I}} \Box (S_2)_{\mathcal{I}}$.

Finally, the least upper bound of all $T^{\vec{a}}$ with respect to $\sqsubseteq$ must be a specialization of $(S_1)_{\mathcal{I}} \Box (S_2)_{\mathcal{I}}$. But this least upper bound is T and the proof is done. □

The case of unbounded choice can be proven similarily to the last case.

Proposition 9 *Let $S' = @y \bullet S$ be a specification on Y and $\mathcal{I}$ a static constraint on X with $Y \subseteq X$. If $T \sqsubseteq S'$ is $\mathcal{I}$-consistent, then $T \sqsubseteq @y \bullet S_{\mathcal{I}}$.* □

Proposition 10 *Let $S = S_1 \boxtimes S_2$ a specification on Y and $\mathcal{I}$ a static constraint on X with $Y \subseteq X$. If $T \sqsubseteq S$ is $\mathcal{I}$ consistent, then*

$$T \sqsubseteq (S_1)_{\mathcal{I}} \Box wp(S_1)(false) \rightarrow (S_2)_{\mathcal{I}} \sqsubseteq (S_1)_{\mathcal{I}} \boxtimes (S_2)_{\mathcal{I}}.$$

Futhermore, $T \sqsubseteq (S_1)_{\mathcal{I}} \Box wlp(S_1)(false) \rightarrow (S_2)_{\mathcal{I}}$ holds.

Proof. We define $T_1 = wp(S_1)^*(true) \rightarrow T$ and $T_2 = wp(S_1)(false) \rightarrow T$. Let φ be an arbitrary Y-formula. Then we have

$$\begin{aligned} w(l)p(T_1 \Box T_2)(\varphi) \quad &\Leftrightarrow \quad (wp(S_1)^*(true) \wedge w(l)p(T)^*(\varphi)) \vee \\ &\qquad (wp(S_1)(false) \wedge w(l)p(T)^*(\varphi)) \\ &\Leftrightarrow \quad (wp(S_1)^*(true) \vee wp(S_1)(false)) \wedge w(l)p(T)^*(\varphi) \\ &\Leftrightarrow \quad w(l)p(T)^*(\varphi) \quad , \end{aligned}$$

that is T and $T_1 \Box T_2$ are semantically equivalent. By assumption $T \sqsubseteq S_1 \boxtimes S_2$ holds, and hence

$$w(l)p(T)^*(\varphi) \quad \Rightarrow \quad w(l)p(S_1)^*(\varphi) \vee (wp(S_1)(false) \wedge w(l)p(S_2)^*(\varphi))$$

is valid, too. Besides, we can proof

$$\begin{aligned} w(l)p(T_1)^*(\varphi) \quad &\Leftrightarrow \quad wp(S_1)^*(true) \wedge w(l)p(T)^*(\varphi) \\ &\Rightarrow \quad (wp(S_1)^*(true) \wedge w(l)p(S_1)^*(\varphi)) \vee \\ &\qquad \underbrace{(wp(S_1)^*(true) \wedge wp(S_1)(false)}_{\Leftrightarrow false} \wedge w(l)p(S_2)^*(\varphi)) \\ &\Rightarrow \quad w(l)p(S_1)(\varphi) \quad . \end{aligned}$$

But this means $T_1 \sqsubseteq S_1$. Even more,

$$\begin{aligned} w(l)p(T_2)^*(\varphi) \quad &\Leftrightarrow \quad wp(S_1)(false) \wedge w(l)p(T)^*(\varphi) \\ &\Rightarrow \quad (wp(S_1)(false) \wedge w(l)p(S_1)^*(\varphi)) \vee \\ &\qquad (wp(S_1)(false) \wedge w(l)p(S_2)^*(\varphi)) \\ &\Rightarrow \quad \underbrace{(wp(S_1)(false) \wedge wp(S_1)^*(true))}_{\Leftrightarrow false} \vee \\ &\qquad (wp(S_1)(false) \wedge w(l)p(S_2)^*(\varphi)) \\ &\Leftrightarrow \quad w(l)p(wp(S_1)(false) \rightarrow S_2)^*(\varphi) \end{aligned}$$

gives us $T_2 \sqsubseteq wp(S_1)(false) \rightarrow S_2$. Herein, the second implication is due to $\neg wp(S_1)^*(true) \Rightarrow \neg wp(S_1)^*(\varphi)$ and $wlp(S_1)^*(\varphi) \Rightarrow wp(S_1)^*(\varphi)$.
As T_1 and T_2 are $\mathcal{I}$-consistent, we have $T_1 \sqsubseteq (S_1)_{\mathcal{I}}$. Due to Proposition 7 and $(S_1)_{\mathcal{I}} \sqsubseteq S_1$, we derive $T_2 \sqsubseteq wp(S_1)(false) \rightarrow (S_2)_{\mathcal{I}} \sqsubseteq wp((S_1)_{\mathcal{I}})(false) \rightarrow (S_2)_{\mathcal{I}}$, i.e., by definition of the predicate transformers

$$\begin{aligned} T_1 \Box T_2 \quad &\sqsubseteq \quad (S_1)_{\mathcal{I}} \Box wp(S_1)(false) \rightarrow (S_2)_{\mathcal{I}} \\ &\sqsubseteq \quad (S_1)_{\mathcal{I}} \Box wp((S_1)_{\mathcal{I}})(false) \rightarrow (S_2)_{\mathcal{I}} \; = \; (S_1)_{\mathcal{I}} \boxtimes (S_2)_{\mathcal{I}}. \end{aligned}$$

This gives the first statement, the second one becomes obvious when we look at $wp(S_1)(false) \Rightarrow wlp(S_1)(false)$. □

C.1 The Case for Sequences

We come now to the case of sequences. Herein, the definition of δ-$\mathcal{I}$-reducedness will become more apparent. But first, we will show the following lemma.

Lemma 9 *Let $S = S_1; S_2$ be a program specification on Y with S_i on $Y_i \subseteq Y$ for $i = 1, 2$. Let $\mathcal{I}$ be a static invariant on $X = \{x_1, \ldots, x_s\}$ with $Y \subseteq X$. Besides, $X - Y_1 - Y_2 = \{y_1, \ldots, y_m\}$, $X - Y_1 = \{y_1, \ldots, y_m, y_{m+1}, \ldots, y_n\}$, $X - Y_2 = \{y_1, \ldots, y_m, x_{l+1}, \ldots, x_k\}$, $Y_1 = \{x_1, \ldots, x_l, x_{l+1}, \ldots, x_k\}$ and $\{x'_1, \ldots, x'_k\}$ a disjoint copy of Y_1 with $Y'_1 \cap Y = \emptyset$. If S is δ-$\mathcal{I}$-reduced and S_1 deterministic, then*

1. *for all states $\vec{a}$ and $\vec{b}$ with $\models_{\vec{a}} \neg\mathcal{I}$, $\models_{\vec{b}} \neg\mathcal{I}$ and $\models_{\vec{a}} wlp(S)^*(\exists y_1, \ldots, y_m.\ \vec{x} = \vec{b})$, for which*

$$\vec{x} = \vec{a} \Rightarrow \{\vec{x}/\vec{x}'\}.(\forall y_1, \ldots, y_n.wlp(S_2)^*(\exists y_1, \ldots, y_m, x_{l+1}, \ldots, x_k.\vec{x} = \vec{b}) \Rightarrow \mathcal{I})$$

   is a δ-constraint for S_1, $\vec{x} = \vec{a} \Rightarrow \{\vec{x}/\vec{x}'\}.\forall y_1, \ldots, y_n.\mathcal{I}$ is a δ-constraint for S.

2. *for all states $\vec{a}$ and $\vec{b}$ with $\models_{\vec{a}} \mathcal{I}$, $\models_{\vec{b}} \mathcal{I}$ and $\models_{\vec{a}} wlp(S)^*(\exists y_1, \ldots, y_m.\ \vec{x} = \vec{b})$, for which*

$$\vec{x} = \vec{a} \Rightarrow \{\vec{x}/\vec{x}'\}.(\forall y_1, \ldots, y_n.wlp(S_2)^*(\exists y_1, \ldots, y_m, x_{l+1}, \ldots, x_k.\vec{x} = \vec{b}) \Rightarrow \neg\mathcal{I})$$

   is a δ-constraint for S_1, $\vec{x} = \vec{a} \Rightarrow \{\vec{x}/\vec{x}'\}.\forall y_1, \ldots, y_n.\neg\mathcal{I}$ is a δ-constraint for S.

Proof. We will show (i) only. The proof for (ii) is completely analogously. Let $\vec{a}$ and $\vec{b}$ be states with $\models_{\vec{a}} \neg\mathcal{I}$, $\models_{\vec{b}} \neg\mathcal{I}$ und $\models_{\vec{a}} wlp(S)^*(\exists y_1, \ldots, y_m.\ \vec{x} = \vec{b})$ and

$$\vec{x} = \vec{a} \Rightarrow \{\vec{x}/\vec{x}'\}.(\forall y_1, \ldots, y_n.wlp(S_2)^*(\exists y_1, \ldots, y_m, x_{l+1}, \ldots, x_k.\vec{x} = \vec{b}) \Rightarrow \mathcal{I}) \quad (*)$$

a δ-constraint for S_1. Then

$$\begin{aligned} \models_{\vec{a}} wlp(S)^*(\exists y_1, \ldots, y_m.\vec{x} = \vec{b}) &\Leftrightarrow \models_{\vec{a}} wlp(S_1)^*(wlp(S_2)^*(\exists y_1, \ldots, y_m.\vec{x} = \vec{b})) \\ &\Rightarrow \models_{\vec{a}} wp(S_1)(wlp(S_2)^*(\exists y_1, \ldots, y_m.\vec{x} = \vec{b})) \\ &\Rightarrow \models_{\vec{a}} wlp(S_1)(wlp(S_2)^*(\exists y_1, \ldots, y_m.\vec{x} = \vec{b})) \end{aligned}$$

holds, using the definition of $wlp(S)$, S_1 being deterministic and the pairing condition. Moreover, we conclude

$$\models \{\vec{x}'/\vec{x}\}.wlp(\{\vec{x}/\vec{x}'\}.S_1)(\vec{x} = \vec{a} \Rightarrow wlp(\{\vec{x}/\vec{x}'\}.S_2)^*(\exists y_1, \ldots, y_m.\{\vec{x}/\vec{x}'\}.\vec{x} = \vec{b})) \quad (**)$$

by definition of $wlp(S_1)$. By definition of a δ-constraint, $(*)$ implies

$$\models \{\vec{x}'/\vec{x}\}.wlp(\{\vec{x}/\vec{x}'\}.S_1)(\vec{x} = \vec{a} \Rightarrow \{\vec{x}/\vec{x}'\}.\forall y_1, \ldots, y_n.wlp(S_2)^*(\exists y_1, \ldots, y_m, x_{l+1}, \ldots, x_k.\vec{x} = \vec{b}) \Rightarrow \mathcal{I}))$$

and together with $(**)$ further

$$\models \{\vec{x}'/\vec{x}\}.wlp(\{\vec{x}/\vec{x}'\}.S_1)(\vec{x} = \vec{a} \Rightarrow \{\vec{x}/\vec{x}'\}.\forall y_1, \ldots, y_n.\mathcal{I}) \quad .$$

Hence, $\vec{x} = \vec{a} \Rightarrow \{\vec{x}/\vec{x}'\}.\forall y_1, \ldots, y_n.\mathcal{I}$ is a δ-constraint for S_1. As S is δ-$\mathcal{I}$-reduced by assumption, $\vec{x} = \vec{a} \Rightarrow \{\vec{x}/\vec{x}'\}.\forall y_1, \ldots, y_n.\mathcal{I}$ is also a δ-constraint for S. □

Proposition 11 *Let $S = S_1; S_2$ be an $\mathcal{I}$-reduced specification on Y with $\mathcal{I}$ being a static constraint on X with $Y \subseteq X$. If $T \sqsubseteq S$ is $\mathcal{I}$-consistent, then $T \sqsubseteq (S_1)_\mathcal{I}; (S_2)_\mathcal{I}$.*

Proof. Without loss in generality we assume that $wp(T)(true) \Leftrightarrow true$ holds. Then it suffices to show $wlp(S_\mathcal{I})^*(\vec{x} = \vec{a}) \Rightarrow wlp((S_1)_\mathcal{I}; (S_2)_\mathcal{I})^*(\vec{x} = \vec{a})$ for all state characterising formulae $\vec{x} = \vec{a}$. Namely,

$$\begin{aligned}
wlp((S_1)_\mathcal{I})(wlp((S_2)_\mathcal{I})(\varphi(\vec{x}))) &\Leftrightarrow wlp((S_1)_\mathcal{I})(wlp((S_2)_\mathcal{I})(\forall \vec{z}.\neg\varphi(\vec{z}) \Rightarrow \vec{x} \neq \vec{z})) \\
&\Leftrightarrow wlp((S_1)_\mathcal{I})(\forall \vec{z}.\neg\varphi(\vec{z}) \Rightarrow wlp((S_2)_\mathcal{I})(\vec{x} \neq \vec{z})) \\
&\Leftrightarrow \forall \vec{z}.\neg\varphi(\vec{z}) \Rightarrow wlp((S_1)_\mathcal{I})(wlp((S_2)_\mathcal{I})(\vec{x} \neq \vec{z})) \\
&\Rightarrow \forall \vec{z}.\neg\varphi(\vec{z}) \Rightarrow wlp(S_\mathcal{I})(\vec{x} \neq \vec{z}) \\
&\Leftrightarrow wlp(S_\mathcal{I})(\forall \vec{z}.\neg\varphi(\vec{z}) \Rightarrow \vec{x} \neq \vec{z}) \\
&\Leftrightarrow wlp(S_\mathcal{I})(\varphi(\vec{x}))
\end{aligned}$$

holds for all X-formulae φ.

As S_1 is the least upper bound of its deterministic branches with respect to $\sqsubseteq$, we can further assume without loss in generality that S_1 is deterministic. Therefore, we are able to use the stronger properties from Lemma 9.

First, we compute both sides of of the implication above using the GCS normal form from Proposition 4. We obtain

$$wlp(S_\mathcal{I})^*(\vec{x} = \vec{a}) \Leftrightarrow \boxed{\begin{array}{c}(\mathcal{I} \wedge \exists \vec{\xi}.wlp(S)^*(\{\vec{y}/\vec{\xi}\}.\mathcal{I} \wedge \{\vec{y}/\vec{\xi}\}.\vec{x} = \vec{a})) \vee \\ (\neg\mathcal{I} \wedge \exists \vec{\xi}.wlp(S)^*(\{\vec{y}/\vec{\xi}\}.\vec{x} = \vec{a}))\end{array}} \qquad (6)$$

as well as

$$\begin{aligned} wlp((S_1)_{\mathcal{I}};(S_2)_{\mathcal{I}})^*(\vec{x}=\vec{a}) \quad &\Leftrightarrow \quad (\mathcal{I} \wedge wlp(S_1)^*(\exists y_1,\ldots,y_n.(\mathcal{I} \wedge \\ &\qquad wlp((S_2)_{\mathcal{I}})^*(\vec{x}=\vec{a})))) \vee (\neg\mathcal{I} \wedge wlp(S_1)^*(\\ &\qquad \exists y_1,\ldots,y_n.wlp((S_2)_{\mathcal{I}})^*(\vec{x}=\vec{a}))) \\ &\Leftrightarrow \quad (\mathcal{I} \wedge wlp(S_1)^*(\exists y_1,\ldots,y_n.(\mathcal{I} \wedge wlp(S_2)^*(\\ &\qquad \exists y_1,\ldots,y_m,x_{l+1},\ldots,x_k(\mathcal{I} \wedge \vec{x}=\vec{a}))))) \vee \\ &\quad (\neg\mathcal{I} \wedge wlp(S_1)^*(\exists y_1,\ldots,y_n.(\mathcal{I} \wedge wlp(S_2)^*(\\ &\qquad \exists y_1,\ldots,y_m,x_{l+1},\ldots,x_k(\mathcal{I} \wedge \vec{x}=\vec{a}))))) \vee \\ &\quad (\neg\mathcal{I} \wedge wlp(S_1)^*(\exists y_1,\ldots,y_n.(\neg\mathcal{I} \wedge \\ &\quad wlp(S_2)^*(\exists y_1,\ldots,y_m,x_{l+1},\ldots,x_k.\vec{x}=\vec{a})))) \end{aligned}$$

and this is equivalent to

$$\boxed{\begin{array}{c} \exists\xi_1,\ldots,\xi_n.\exists\xi'_1,\ldots,\xi'_m,\xi'_{l+1},\ldots,\xi'_k.(wlp(S_1)^*(\{\vec{y}/\vec{\xi}\}.\mathcal{I} \wedge \\ \{\vec{y}/\vec{\xi}\}.wlp(S_2)^*(\{\vec{y}/\vec{\xi}'\}.(\mathcal{I} \wedge \vec{x}=\vec{a})))) \vee \\ \exists\xi_1,\ldots,\xi_n.\exists\xi'_1,\ldots,\xi'_m,\xi'_{l+1},\ldots,\xi'_k.\neg\mathcal{I} \wedge \\ (wlp(S_1)^*(\neg\{\vec{y}/\vec{\xi}\}.\mathcal{I} \wedge \{\vec{y}/\vec{\xi}\}.wlp(S_2)^*(\{\vec{y}/\vec{\xi}'\}.\vec{x}=\vec{a}))) \end{array}} \tag{7}$$

Case 1. We assume $\vec{x}=\vec{a} \Rightarrow \neg\mathcal{I}$. Then $wlp(S_{\mathcal{I}})^*(\vec{x}=\vec{a}) \Rightarrow wlp(S_{\mathcal{I}})^*(\neg\mathcal{I}) \Rightarrow \neg\mathcal{I}$ follows as $S_{\mathcal{I}}$ is $\mathcal{I}$-consistent. Since we also $wlp(S_{\mathcal{I}})^*(\vec{x}=\vec{a})$ assume, we look at the second line of formula (6). We show, that we can derive the second subformula of (7). Assuming consistency, we are allowed to neglect $\neg\mathcal{I}$, i.e., we need to derive

$$\begin{aligned} &\exists\xi_1,\ldots,\xi_n.\exists\xi'_1,\ldots,\xi'_m,\xi'_{l+1},\ldots,\xi'_k.(\neg\mathcal{I} \wedge \\ &\quad (wlp(S_1)^*(\neg\{\vec{y}/\vec{\xi}\}.\mathcal{I} \wedge \{\vec{y}/\vec{\xi}\}.wlp(S_2)^*(\{\vec{y}/\vec{\xi}'\}.\vec{x}=\vec{a})))) \end{aligned} \tag{8}$$

Suppose, (8) does not hold. Then, there is a state $\vec{b}$ with

$$\begin{aligned} \models_{\vec{b}} wlp(S_1)(\forall\xi_1,\ldots,\xi_n.\{\vec{y}/\vec{\xi}\}.(wlp(S_2)^* \\ (\exists\xi'_1,\ldots,\xi'_m,\xi'_{l+1},\ldots,\xi'_k.\{\vec{y}/\vec{\xi}'\}.\vec{x}=\vec{a}) \Rightarrow \mathcal{I}))). \end{aligned} \tag{9}$$

We compute that (9) is equivalent to

$$\models_{\vec{b}} \{\vec{x}'/\vec{x}\}.wlp(\{\vec{x}/\vec{x}'\}.S_1)(\{\vec{x}/\vec{x}'\}.$$
$$\underbrace{\forall\xi_1,\ldots,\xi_n.\{\vec{y}/\vec{\xi}\}.(wlp(S_2)^*(\exists\xi'_1,\ldots,\xi'_m,\xi'_{l+1},\ldots,\xi'_k.\{\vec{y}/\vec{\xi}'\}.\vec{x}=\vec{a}) \Rightarrow \mathcal{I}))}_{R}$$

and therefore to

$$\models_{\vec{b}} \{\vec{x}'/\vec{x}\}.(\forall\vec{x}''.wlp(\{\vec{x}/\vec{x}'\}.S_1)^*(\vec{x}'=\vec{x}'') \Rightarrow \{\vec{x}'/\vec{x}''\}.\{\vec{x}/\vec{x}'\}.R)$$

applying Lemma 1 to $wlp(\{\vec{x}/\vec{x}'\}.S_1)(\{\vec{x}/\vec{x}'\}.R)$. From this, we derive the equivalence to

$$\vec{x} = \vec{b} \Rightarrow \{\vec{x}'/\vec{x}\}.(\forall \vec{x}''.wlp(\{\vec{x}/\vec{x}'\}.S_1)^*(\vec{x}' = \vec{x}'') \Rightarrow \{\vec{x}'/\vec{x}''\}.\{\vec{x}/\vec{x}'\}.R) \Leftrightarrow$$
$$\{\vec{x}'/\vec{x}\}.(\forall \vec{x}''.wlp(\{\vec{x}/\vec{x}'\}.S_1)^*(\vec{x}' = \vec{x}'') \Rightarrow (\vec{x} = \vec{b} \Rightarrow \{\vec{x}'/\vec{x}''\}.\{\vec{x}/\vec{x}'\}.R)) \Leftrightarrow$$
$$\{\vec{x}'/\vec{x}\}.wlp(\{\vec{x}/\vec{x}'\}.S_1)(\vec{x} = \vec{b} \Rightarrow \{\vec{x}/\vec{x}'\}.R).$$

But then

$$\vec{x} = \vec{b} \Rightarrow \{\vec{x}/\vec{x}'\}.(\forall \xi_1, \ldots, \xi_n.$$
$$\{\vec{y}/\vec{\xi}\}.(wlp(S_2)^*(\exists \xi_1', \ldots, \xi_m', \xi_{l+1}', \ldots, \xi_k'.\{\vec{y}/\vec{\xi}'\}.\vec{x} = \vec{a}) \Rightarrow \mathcal{I})) \qquad (10)$$

is a δ-constraint for S_1. As not only $\models_{\vec{b}} \neg\mathcal{I}$, but also $\models_{\vec{a}} \neg\mathcal{I}$ is valid, Lemma 9 (i) implies that

$$\vec{x} = \vec{b} \Rightarrow \{\vec{x}/\vec{x}'\}.(\forall \xi_1, \ldots, \xi_n.\{\vec{y}/\vec{\xi}\}.\mathcal{I}) \qquad (11)$$

is a δ-constraint for S. We conclude

$$\{\vec{x}'/\vec{x}\}.wlp(\{\vec{x}/\vec{x}'\}.S)(\vec{x} = \vec{b} \Rightarrow \{\vec{x}/\vec{x}'\}.(\forall \xi_1, \ldots, \xi_n.\{\vec{y}/\vec{\xi}\}.\mathcal{I}))$$

and this is equivalent to

$$\models_{\vec{b}} \{\vec{x}'/\vec{x}\}.wlp(\{\vec{x}/\vec{x}'\}.S)(\{\vec{x}/\vec{x}'\}.(\forall \xi_1, \ldots, \xi_n.\{\vec{y}/\vec{\xi}\}.\mathcal{I})) \Leftrightarrow$$
$$\models_{\vec{b}} wlp(S)(\forall y_1, \ldots, y_n.\mathcal{I}) \qquad (12)$$

following a similar computation as above. On the other hand, we apply monotonicity on the assumption $\vec{x} = \vec{a} \Rightarrow \neg\mathcal{I}$ and use $S_{\mathcal{I}} \sqsubseteq S$ to compute

$$\begin{array}{lll} wlp(S_{\mathcal{I}})^*(\vec{x} = \vec{a}) & \Rightarrow & wlp(S_{\mathcal{I}})^*(\neg\mathcal{I}) \\ & \Rightarrow & wlp(S)^*(\neg\mathcal{I}) \\ & \Rightarrow & wlp(S)^*(\exists y_1, \ldots, y_n.\neg\mathcal{I}) \end{array}.$$

But this is a contradiction since

$$wlp(S)^*(\exists y_1, \ldots, y_n.\neg\mathcal{I}) \quad \Leftrightarrow \quad \neg wlp(S)(\forall y_1, \ldots, y_n.\mathcal{I})$$

holds.

Case 2. Now we assume $\vec{x} = \vec{a} \Rightarrow \mathcal{I}$ and $\models_{\vec{b}} wlp(S_{\mathcal{I}})^*(\vec{x} = \vec{a})$. Following (6) we distinguish further.

Case 2.1. We suppose $\models_{\vec{b}} \neg\mathcal{I} \wedge \exists\vec{\xi}.wlp(S)^*(\{\vec{y}/\vec{\xi}\}.\vec{x} = \vec{a})$. For state $\vec{b}$ we have

$$\exists\vec{\xi}.wlp(S_1)^*(wlp(S_2)^*(\{\vec{y}/\vec{\xi}\}.\vec{x} = \vec{a})) \Leftrightarrow$$
$$\exists\vec{\xi}.wlp(S_1)^*((\mathcal{I} \vee \neg\mathcal{I}) \wedge wlp(S_2)^*(\{\vec{y}/\vec{\xi}\}.\vec{x} = \vec{a})) \Leftrightarrow$$
$$\exists\vec{\xi}.wlp(S_1)^*(\mathcal{I} \wedge wlp(S_2)^*(\{\vec{y}/\vec{\xi}\}.(\vec{x} = \vec{a} \wedge \mathcal{I}))) \vee$$
$$\exists\vec{\xi}.wlp(S_1)^*(\neg\mathcal{I} \wedge wlp(S_2)^*(\{\vec{y}/\vec{\xi}\}.\vec{x} = \vec{a}))$$

and therefore (7). This gives the proof of case 2.1.
Case 2.2. We suppose $\models_{\vec{b}} \mathcal{I} \wedge \exists\vec{\xi}.wlp(S)^*(\{\vec{y}/\vec{\xi}\}.(\mathcal{I} \wedge \vec{x} = \vec{a}))$ and show that

$$\models_{\vec{b}} \exists\xi_1,\ldots,\xi_n.\exists\xi'_1,\ldots,\xi'_m,\xi'_{l+1},\ldots,\xi'_k.(wlp(S_1)^*(\{\vec{y}/\vec{\xi}\}.\mathcal{I} \wedge \{\vec{y}/\vec{\xi}\}.wlp(S_2)^*(\{\vec{y}/\vec{\xi}'\}.(\mathcal{I} \wedge \vec{x} = \vec{a})))) \tag{13}$$

follows. This implies the first subformula in (7). According to case 1, we assume that (13) does not hold. Similar to the computations above, we conclude that

$$\vec{x} = \vec{b} \Rightarrow \{\vec{x}/\vec{x}'\}.(\forall\xi_1,\ldots,\xi_n.\{\vec{y}/\vec{\xi}\}.(wlp(S_2)^*(\exists\xi'_1,\ldots,\xi'_m,\xi'_{l+1},\ldots,\xi'_k.\{\vec{y}/\xi'\}.\vec{x} = \vec{a}) \Rightarrow \neg\mathcal{I}))$$

is a δ-constraint for S_1. Using Lemma 9 (ii) as well as $\models_{\vec{b}} \mathcal{I}$ and $\models_{\vec{a}} \mathcal{I}$, we can conclude that

$$\vec{x} = \vec{b} \Rightarrow \{\vec{x}/\vec{x}'\}.(\forall\xi_1,\ldots,\xi_n.\{\vec{y}/\vec{\xi}\}.\neg\mathcal{I})$$

is a δ-constraint for S. We derive

$$\{\vec{x}'/\vec{x}\}.wlp(\{\vec{x}/\vec{x}'\}.S)(\vec{x} = \vec{b} \Rightarrow \{\vec{x}/\vec{x}'\}.(\forall\xi_1,\ldots,\xi_n.\{\vec{y}/\vec{\xi}\}.\neg\mathcal{I}))$$

and further

$$\models_{\vec{b}} \{\vec{x}'/\vec{x}\}.wlp(\{\vec{x}/\vec{x}'\}.S)(\{\vec{x}/\vec{x}'\}.(\forall\xi_1,\ldots,\xi_n.\{\vec{y}/\vec{\xi}\}.\neg\mathcal{I})) \quad ,$$

i.e., equivalence to

$$\models_{\vec{b}} wlp(S)(\forall y_1,\ldots,y_n.\neg\mathcal{I}). \tag{14}$$

Due to our assumptions and $\vec{x} = \vec{a} \Rightarrow \mathcal{I}$ we can also conclude that

$$\begin{aligned} wlp(S_{\mathcal{I}})^*(\vec{x} = \vec{a}) &\Rightarrow wlp(S_{\mathcal{I}})^*(\mathcal{I}) \\ &\Rightarrow wlp(S)^*(\mathcal{I}) \\ &\Rightarrow wlp(S)^*(\exists y_1,\ldots,y_n.\mathcal{I}) \\ &\Leftrightarrow \neg wlp(S)(\forall y_1,\ldots,y_n.\neg\mathcal{I}) \end{aligned}$$

holds, a contradiction to (14). This gives the proof for case 2.2. □

C.2 The Recursive Case

In this appendix we prove the upper bound theorem for recursive operations restricted to simple WHILE-loops in the form of $f(S) = \mathcal{P} \to T; S\Box\neg\mathcal{P} \to skip$ for which we know the existence of least fixpoints according to subsection 3.2. For this we need some additional lemmata.

For recursive guarded commands the monotonicity of all operation constructors with respect to the Nelson-order $\preceq$ is fundamental [4]. Unfortunately, a similiar result does not hold for the specialization order $\sqsubseteq$. More precisely, the result is false for the $\boxtimes$-constructor in its first component.

Lemma 10 *Let $f(S)$ be a guarded-command expression with the program variable S in which restricted choice $\boxtimes$ does not occur. Then f is monotonic with respect to the specialization order $\sqsubseteq$.*

Proof. The proof is done by structural induction. For each constructor it is completely analogous to the corresponding proof for the Nelson-order in [4]. We omit the details. □

In [6, Proposition 20, p.120] we have seen that $S'_{\mathcal{I}}$ may contain the choice-constructor instead of restricted choice, provided we include some guard. Replacing within a recursive operation some $S_1 \boxtimes S_2$ by $(S_1)_{\mathcal{I}} \boxtimes (S_2)_{\mathcal{I}}$ would destroy the required result.

The next lemma follows from taking together the cases in the upper bound theorem for preconditionings $\rightarrow$, choices $\Box$, unbounded choices @ and restricted choices $\boxtimes$.

Lemma 11 *Let T be a consistent specialization of some $\mathcal{I}$-reduced $f(S')$ with respect to $\mathcal{I}$, where $f(S)$ is an expression built from the constructors of guarded commands. Construct $f_{\mathcal{I}}(S)$ from $f(S)$ as follows:*

(i) Each restricted choice $S_1 \boxtimes S_2$ occuring within $f(S)$ will be replaced by $S_1 \Box wlp(S_1)(false) \rightarrow S_2$.

(ii) Then each basic operation, i.e. skip and assignments will be replaced by their GCSs with respect to $\mathcal{I}$.

Then we have $T \sqsubseteq f_{\mathcal{I}}(S'_{\mathcal{I}})$. □

We must now face the main difficulty to bring together two different partial orders, namely the specialization order $\sqsubseteq$ which is fundamental for GCSs and the Nelson-order $\preceq$ required for recursion.

In order to accomplish this we will need to make use of another limit operator $\lim_{i\in\mathbb{N}} f^i(loop)_{\mathcal{I}}$. Semantics is completely analogously assigned as for the case of $\lim_{i\in\mathbb{N}} f^i(loop)$. Therefore, we receive a corresponding result to Lemma 2 which can be obtained by using Proposition 4. It then is straightforward to verify counterparts for Lemma 3 and Lemma 4, finally.

Lemma 12 *Let $\mathcal{I}$ be a static constraint and $f(T) = \mathcal{P} \rightarrow S; T \Box \neg\mathcal{P} \rightarrow skip$ such that T does not occur within S. Then for each $j \in \mathbb{N}$, there exist predicate transformers $\tau_l^{\mathcal{I}}(j)$ and $\tau^{\mathcal{I}}(j)$ on arithmetic predicates such that the following properties are satisfied:*

(i) for each arithmetic predicate $\varphi(\vec{x})$, the results of applying these predicate transformers are arithmetic predicates in i and x, say

$$\chi_j^{1,\mathcal{I}}(i,\vec{x}) = \tau_l^{\mathcal{I}}(j)(\varphi(\vec{x})) \qquad \text{and} \qquad \chi_j^{2,\mathcal{I}}(i,\vec{x}) = \tau^{\mathcal{I}}(j)(\varphi(\vec{x}))$$

(ii) for $j = h(\varphi)$ we obtain

$$\forall \vec{x}.\forall i.\left(\chi_j^{1,\mathcal{I}}(i,\vec{x}) \Leftrightarrow wlp(f^i(loop))(\varphi(\vec{x}))\right) \quad \text{and}$$
$$\forall \vec{x}.\forall i.\left(\chi_j^{2,\mathcal{I}}(i,\vec{x}) \Leftrightarrow wp(f^i(loop))(\varphi(\vec{x}))\right)$$

with $\vec{x} = x_{i_1},\ldots,x_{i_k}$.

Proof. We follow closely the proof Lemma of 2 where we obtained a primitve recursive function $\bar{g}$ such that

$$Q_1'(\bar{g}(k),j,\vec{x}) \quad = \quad wlp(f^k(loop))(\varphi(\vec{x}))$$

is satisfied. Herein, $\bar{g}(k)$ gives us the Gödel number of $f^k(loop)$. Using the normal form for GCSs from Proposition 4, we can easily derive a further primitive recursive function s such that the composition of s with $\bar{g}$ yields the Gödel number $(s \circ \bar{g})(k)$ for $f^k(loop)_{\mathcal{I}}$. Notice, that $loop_{\mathcal{I}} = loop$ holds. In particular, we obtain

$$q_1^{\mathcal{I}}((s \circ \bar{g})(k),j,\vec{x}) \quad = \quad wlp(f^k(loop)_{\mathcal{I}})(\varphi(\vec{x})).$$

Then, we define predicates $Q_1^{\mathcal{I}}(k,j,\vec{x}) = q_1^{\mathcal{I}}((s \circ \bar{g})(k),j,\vec{x})$ and an extension of $\bar{Q}^{\mathcal{I}}(h(\psi),h(\varphi),\vec{x}) \Leftrightarrow ((\mathcal{P} \Rightarrow wlp(S)(\psi)) \wedge (\neg\mathcal{P} \Rightarrow \varphi))$ with $\psi,\varphi \in \mathbb{F}$. We conclude

$$Q_1^{\mathcal{I}}(k,j,\vec{x}) \quad = \quad q_1^{\mathcal{I}}((s \circ \bar{g})(k),j,\vec{x}) \quad = \quad \bar{Q}^{\mathcal{I}}(h(\psi),h(\varphi),\vec{x}) \quad ,$$

where $h(\varphi) = j$ and $\psi(\vec{x}) = wlp(f^{k-1}(loop)_{\mathcal{I}})(\varphi(\vec{x})) = Q_1^{\mathcal{I}}(k-1,j,\vec{x})$. In summary, we receive

$$\begin{array}{rcl} Q_1^{\mathcal{I}}(0,j,\vec{x}) & = & true \quad \text{and} \\ Q_1^{\mathcal{I}}(k+1,j,\vec{x}) & = & \bar{Q}^{\mathcal{I}}(h(Q_1^{\mathcal{I}}(k,j,\vec{x})),j,\vec{x}). \end{array}$$

Now we take $\tau_l^{\mathcal{I}}(j)(\varphi(\vec{x})) = \chi_j^{1,\mathcal{I}}(k,\vec{x}) = Q_1^{\mathcal{I}}(k,j,\vec{x})$ and conlcude as in Lemma 2. □

We now define *limit operators* $\lim_{i\in\mathbb{N}} f^i(loop)_{\mathcal{I}}$ with help of the predicate transformers $\tau_l^{\mathcal{I}}(j)$ and $\tau^{\mathcal{I}}(j)$:

$$wlp\left(\lim_{i\in\mathbb{N}} f^i(loop)_{\mathcal{I}}\right)(\varphi(\vec{x})) \quad \Leftrightarrow \quad \forall i.\chi_{h(\varphi)}^{1,\mathcal{I}}(i,\vec{x}) \qquad \text{and}$$
$$wp\left(\lim_{i\in\mathbb{N}} f^i(loop)_{\mathcal{I}}\right)(\varphi(\vec{x})) \quad \Leftrightarrow \quad \exists i.\chi_{h(\varphi)}^{2,\mathcal{I}}(i,\vec{x})$$

for $\chi_{h(\varphi)}^{1,\mathcal{I}}(i,\vec{x}) = \tau_l^{\mathcal{I}}(h(\varphi))(\varphi(\vec{x}))$ and $\chi_{h(\varphi)}^{2,\mathcal{I}}(i,\vec{x}) = \tau^{\mathcal{I}}(h(\varphi))(\varphi(\vec{x}))$.

Lemma 13 *The definition of limits $\lim_{i\in\mathbb{N}} f^i(loop)_{\mathcal{I}}$ is sound.*

Proof. The proof follows exactly the one from Lemma 3. We just need to mention that $\{f^i(loop)_{\mathcal{I}} \mid i \in \mathbb{N}\}$ is a chain with respect to the Nelson-order $\preceq$. As $loop_{\mathcal{I}} =$

loop is the $\preceq$-minimum, we have $loop_{\mathcal{I}} \preceq f(loop)_{\mathcal{I}}$. Since $\{f^i(loop) \mid i \in \mathbb{N}\}$ is a $\preceq$-chain and therefore $f^i(loop) \preceq f^{i+1}(loop)$ holds, we can finally derive $f^i(loop)_{\mathcal{I}} \preceq f^{i+1}(loop)_{\mathcal{I}}$ (see also Lemma 15). □

The following lemma gives us the corresponding result to Lemma 4. The proof is again completely analogous.

Lemma 14 *The chain* $\{f^i(loop)_{\mathcal{I}} \mid i \in \mathbb{N}\}$ *has a least upper bound, namely* $\lim_{i\in\mathbb{N}} f^i(loop)_{\mathcal{I}}$. □

We are now prepared to bring specialization- and Nelson-order together.

Lemma 15 *Let T und S be Y-operations. Furthermore, let $\mathcal{I}$ be an invariant on X for $Y \subseteq X$. Then we have:*

(i) If $T \preceq S$ holds, then $T_{\mathcal{I}} \preceq S_{\mathcal{I}}$ follows.

(ii) $\left(\lim_{i\in\mathbb{N}} f^i(loop)\right)_{\mathcal{I}} \sqsubseteq \lim_{i\in\mathbb{N}}(f^i(loop))_{\mathcal{I}}$.

Proof. (i) Here we use the normal form of a GCS given in Proposition 4. The first result follows immediately, because all constructors are monotonic in the Nelson-order $\preceq$.

(ii) First, $\lim_{i\in\mathbb{N}} f^i(loop)$ is the least upper bound of $\{f^i(loop) \mid i \in \mathbb{N}\}$ with respect to the Nelson-order according to Lemma 4, i.e. especially $f^i(loop) \preceq \lim_{i\in\mathbb{N}} f^i(loop)$ holds for arbitrary $i \in \mathbb{N}$. From this and (i) we get $f^i(loop)_{\mathcal{I}} \preceq \left(\lim_{i\in\mathbb{N}} f^i(loop)\right)_{\mathcal{I}}$, i.e. $\left(\lim_{i\in\mathbb{N}} f^i(loop)\right)_{\mathcal{I}}$ is an upper bound for $\{f^i(loop)_{\mathcal{I}} \mid i \in \mathbb{N}\}$. Using Lemma 14, $\lim_{i\in\mathbb{N}} f^i(loop)_{\mathcal{I}}$ is the least upper bound of the chain $\{f^i(loop)_{\mathcal{I}} \mid i \in \mathbb{N}\}$ which means that $\lim_{i\in\mathbb{N}} f^i(loop)_{\mathcal{I}} \preceq \left(\lim_{i\in\mathbb{N}} f^i(loop)\right)_{\mathcal{I}}$ must hold. Therefore, we receive

$$wp\left(\lim_{i\in\mathbb{N}} f^i(loop)_{\mathcal{I}}\right)(\varphi) \quad \Rightarrow \quad wp\left(\left(\lim_{i\in\mathbb{N}} f^i(loop)\right)_{\mathcal{I}}\right)(\varphi)$$

according to the definition of the Nelson-order.

Once again we make use of Proposition 4 in order to compute

$$\begin{aligned}
wlp\left(\lim_{i\in\mathbb{N}} f^i(loop)_{\mathcal{I}}\right)(\varphi) &\Leftrightarrow\\
\forall i.i \in \mathbb{N} \Rightarrow wlp\left(f^i(loop)_{\mathcal{I}}\right)(\varphi) &\Leftrightarrow\\
\forall i.i \in \mathbb{N} \Rightarrow wlp((\mathcal{I} \to f^i(loop); @\vec{z}' \bullet \vec{z} := \vec{z}'; \mathcal{I} \to skip)\boxtimes &\\
(\neg\mathcal{I} \to f^i(loop); @\vec{z}' \bullet \vec{z} := \vec{z}'))(\varphi) &\Leftrightarrow\\
\forall i.i \in \mathbb{N} \Rightarrow ((\mathcal{I} \Rightarrow wlp(f^i(loop))(\forall \vec{z}'.\{\vec{z}/\vec{z}'\}.\mathcal{I} \Rightarrow \varphi)) \wedge &\\
(\neg\mathcal{I} \Rightarrow wlp(f^i(loop))(\forall \vec{z}'.\{\vec{z}/\vec{z}'\}.\varphi))) &\Leftrightarrow\\
(\mathcal{I} \Rightarrow \forall i.i \in \mathbb{N} \Rightarrow wlp(f^i(loop))(\forall \vec{z}'.\{\vec{z}/\vec{z}'\}.\mathcal{I} \Rightarrow \varphi)) \wedge &\\
(\neg\mathcal{I} \Rightarrow \forall i.i \in \mathbb{N} \Rightarrow wlp(f^i(loop))(\forall \vec{z}'.\{\vec{z}/\vec{z}'\}.\varphi)) &\Leftrightarrow\\
(\mathcal{I} \Rightarrow wlp\left(\lim_{i\in\mathbb{N}} f^i(loop)\right)(\forall \vec{z}'.\{\vec{z}/\vec{z}'\}.\mathcal{I} \Rightarrow \varphi)) \wedge &\\
(\neg\mathcal{I} \Rightarrow wlp\left(\lim_{i\in\mathbb{N}} f^i(loop)\right)(\forall \vec{z}'.\{\vec{z}/\vec{z}'\}.\varphi)) &\Leftrightarrow\\
wlp((\mathcal{I} \to \lim_{i\in\mathbb{N}} f^i(loop); @\vec{z}' \bullet \vec{z} := \vec{z}'; \mathcal{I} \to skip)\boxtimes &\\
(\neg\mathcal{I} \to \lim_{i\in\mathbb{N}} f^i(loop); @\vec{z}' \bullet \vec{z} := \vec{z}'))(\varphi) &\Leftrightarrow\\
wlp\left(\left(\lim_{i\in\mathbb{N}} f^i(loop)\right)_{\mathcal{I}}\right)(\varphi) \quad , &
\end{aligned}$$

i.e.

$$wlp\left(\lim_{i\in\mathbb{N}} f^i(loop)_{\mathcal{I}}\right)(\varphi) \Rightarrow wlp\left(\left(\lim_{i\in\mathbb{N}} f^i(loop)\right)_{\mathcal{I}}\right)(\varphi) \quad ,$$

supplies the asserted specialization. □

We are now able to give the main proof.

Proposition 12 Let $S' = \mu T_j.f(T_j)$ with $f(T_j) = \mathcal{P} \to T; T_j \square \neg\mathcal{P} \to skip$ be an $\mathcal{I}$-reduced Y-operation and $T \sqsubseteq S'$ a consistent specialization with respect to some X-invariant $\mathcal{I}$ with $Y \subseteq X$. Then we have $T \sqsubseteq \mu T_j.f_{\mathcal{I}}(T_j)$, where $f_{\mathcal{I}}(T_j)$ is built as in Lemma 11.

Proof. Since S' is a fixpoint we have $S' = f(S')$. T is an $\mathcal{I}$-reduced consistent specialization of S' by assumption, so the specialization

$$T \sqsubseteq f_{\mathcal{I}}(S'_{\mathcal{I}}) = f_{\mathcal{I}}\left(\left(\lim_{i\in\mathbb{N}} f^i(loop)\right)_{\mathcal{I}}\right)$$

follows by Lemma 11. Due to the monotonicity of $f_{\mathcal{I}}$ and because of Lemma 15 (ii) we derive further

$$f_{\mathcal{I}}\left(\left(\lim_{i\in\mathbb{N}} f^i(loop)\right)_{\mathcal{I}}\right) \sqsubseteq f_{\mathcal{I}}\left(\underbrace{\lim_{i\in\mathbb{N}} \overbrace{\left(f^i(loop)\right)_{\mathcal{I}}}^{T_{1i}}}_{T_1}\right) .$$

We set $T_{2i} = f^i_{\mathcal{I}}(loop)$ and show $T_{1i} \sqsubseteq T_{2i}$ for all $i \in \mathbb{N}$ by induction. The case $i = 0$ gives $T_{10} = loop_{\mathcal{I}} = loop = T_{20}$. In the case $i > 0$ we can assume $T_{1j} \sqsubseteq T_{2j}$ for all $j < i$. T_{1i} is an $\mathcal{I}$-consistent specialization of $f^i(loop) = f(f^{i-1}(loop))$, hence we conclude

$$T_{1i} \sqsubseteq f_{\mathcal{I}}\left(\left(f^{i-1}(loop)\right)_{\mathcal{I}}\right) = f_{\mathcal{I}}\left(T_{1(i-1)}\right).$$

by Lemma 11. Now, we apply the induction hypothesis and the monotonicity of $f_{\mathcal{I}}$ in order to obtain $f_{\mathcal{I}}\left(T_{1(i-1)}\right) \sqsubseteq f_{\mathcal{I}}\left(T_{2(i-1)}\right) = T_{2i}$, i.e. together $T_{1i} \sqsubseteq T_{2i}$ as asserted.
For $T_2 = \lim_{i\in\mathbb{N}} f^i_{\mathcal{I}}(loop)$ follows

$$\begin{aligned} wlp(T_2)(\varphi) &\Leftrightarrow \forall i.i \in \mathbb{N} \Rightarrow wlp(T_{2i})(\varphi) \\ &\Rightarrow \forall i.i \in \mathbb{N} \Rightarrow wlp(T_{1i})(\varphi) \\ &\Leftrightarrow wlp(T_1)(\varphi) \end{aligned}$$

and

$$\begin{aligned} wp(T_2)(\varphi) &\Leftrightarrow \exists i.i \in \mathbb{N} \wedge wp(T_{2i})(\varphi) \\ &\Rightarrow \exists i.i \in \mathbb{N} \wedge wp(T_{1i})(\varphi) \\ &\Leftrightarrow wp(T_1)(\varphi) \quad , \end{aligned}$$

thus the specialization $T_1 \sqsubseteq T_2$. Finally, we receive by applying Lemma 10

$$T \sqsubseteq f_{\mathcal{I}}(T_1) \sqsubseteq f_{\mathcal{I}}(T_2) = T_2 = \mu T_j . f_{\mathcal{I}}(T_j) \quad ,$$

where we use the fact that T_2 is a fixpoint. □

References

[1] J. Bell, M. Machover. *A Course in Mathematical Logic*. North-Holland 1977.

[2] E. Börger, E. Grädel, Y. Gurevich. *The Classical Decision Problem*. Springer 1997.

[3] S. Link. *Eine Theorie der Konsistenzerzwingung auf der Basis arithmetischer Logik*. M.Sc. Thesis (in German). TU Clausthal 2000.

[4] G. Nelson. A Generalization of Dijkstra's Calculus. *ACM TOPLAS*. vol. 11 (4): 517-561. 1989.

[5] K.-D. Schewe. Fundamentals of Consistency Enforcement. In H. Jaakkola, H. Kangassalo, E. Kawaguchi (eds.). *Information Modelling and Knowledge Bases* X: 275-291. IOS Press 1999.

[6] K.-D. Schewe, B. Thalheim. Towards a Theory of Consistency Enforcement. *Acta Informatica*. vol. 36: 97-141. 1999.

[7] K.-D. Schewe, B. Thalheim. Limitations of Rule Triggering Systems for Integrity Maintenance in the Context of Transition Specifications. *Acta Cybernetica*. vol. 13: 277-304. 1998.

[8] K.-D. Schewe, B. Thalheim, J. Schmidt, I. Wetzel. Integrity Enforcement in Object Oriented Databases. In U. Lipeck, B. Thalheim (eds.). *Modelling Database Dynamics*: 174-195. Workshops in Computing. Springer 1993.

[9] B. Thalheim. *Dependencies in Relational Databases*. Teubner 1991.

Received May, 2000

Acta Cybernetica 15 (2002) 417–455.

Definition of a Parallel Execution Model with Abstract State Machines*

Zsolt Németh†

Abstract

Languages, architectures and execution models are strongly related. A new architectural platform makes necessary to modify the execution model in order to exploit all the advantages of the underlying architecture while preserving its main characteristics. The latter issue requires a careful analysis of the design process. Abstract State Machines offer a powerful method for aiding complex system design. In this paper some aspects of its application are presented by taking the redesign process of a parallel Prolog model as an example.

1 Introduction

The research work presented in this paper aimed at the design of a Prolog interpreter on a multithreaded architecture. However, the certain project represents just the framework and the goal is more general: investigating how a dataflow based model fits a kind of hybrid multithreaded architecture and what the conditions of efficient work are. In a wider scope it deals with the relationship of computational models and the underlying physical architecture.

LOGFLOW is a fine-grained all-solution parallel (reduced) Prolog system for distributed memory architectures. Its abstract execution model called Logicflow [13] can be considered as a sort of macro dataflow scheme, whereas its abstract machine model is the Distributed Data Driven Prolog Abstract Machine [14] (3DPAM). 3DPAM tries to make a connection between a dataflow based execution model and a kind of von Neumann physical architecture.

A hybrid multithreaded platform offers the possibility of creating a more efficient Prolog abstract machine. Its ability to hide latencies due to remote memory access or synchronisation (multithreading) opens a new way for representing Prolog data (heap) and managing the variables. On the other hand, its hybrid feature, i.e. support for both the fast sequential and dataflow execution, is close to the

*This research was supported by the National Research Grant (OTKA) registered under No. T-022106. Presented at the Conference of PhD Students in Computer Science, July 20-23, 2000, Szeged.

†MTA SZTAKI Computer and Automation Research Institute of the Hungarian Academy of Sciences, Budapest, P.O.Box 63. H-1518, E-mail: zsnemeth@sztaki.hu

macro dataflow model of LOGFLOW and makes possible an efficient realisation of dataflow nodes and token flows. To exploit the latter property at the abstract machine level, a new abstract execution model is necessary, too. The new execution model has been derived from the Logicflow in three major steps by changing the way how solution streams are separated, the way how solutions are propagated and by grouping together elementary nodes [18]. Whereas the gain in efficiency is obvious (qualitatively), it is not the case for correctness and semantical equivalence of the models.

The work presented in the paper is a study on the application of a formal method called Abstract State Machines in proving the correctness of the redesign. Abstract State Machines (Gurevich's ASMs, formerly known as evolving algebras) offer a way for the design and analysis of complex hardware and software systems [3] [9]. They are similar to Turing machines in a sense that they simulate algorithms yet, they are able to describe semantics at arbitrary levels of abstraction. An ASM consists of a finite set of transition rules by which the system is driven form state to state, each represented by sets with relations and functions (algebras). By refinement steps a "more abstract" model can be turned into a "more concrete" one and by relating their states and transition rules (by proof mapping) their relative correctness and completeness can be proven. In several refinement steps the equivalence of the models can be shown.

The refinement technique is applied at deriving the new execution model via a series of submodels. LOGFLOW is modeled as an ASM and modifications are introduced by successive new ASMs where each modification step can be checked. Furthermore, implementation steps, creating an interpreter engine can be conducted and checked in the same way.

In Section 2 the notion of computational models are introduced and the circumstances are explained why the modification of Logicflow became necessary. It also summarises the main steps of redesign. Section 3 is a brief introduction to ASMs and their applications. Section 4 puts the design into the framework of ASMs: the initial model and the first derivation are introduced. Finally, in Section 5 the correctness of the first modification is shown.

2 Computational models

Computational models are considered as a higher level of abstraction above languages and architectures [22]. In the course of LOGFLOW project a highly abstract, dataflow based parallel and distributed model called Logicflow [13] has been derived from Prolog language (Figure 1.a). Target architectures were represented of parallel von Neumann types, primarily transputers and networks of workstations. The abstract execution model cannot be implemented directly on the physical machine model but a virtual machine, the so called abstract machine layer is introduced between the execution model and the physical machine model. This way of execution via abstract interpretation is general in case of Prolog and declarative languages.

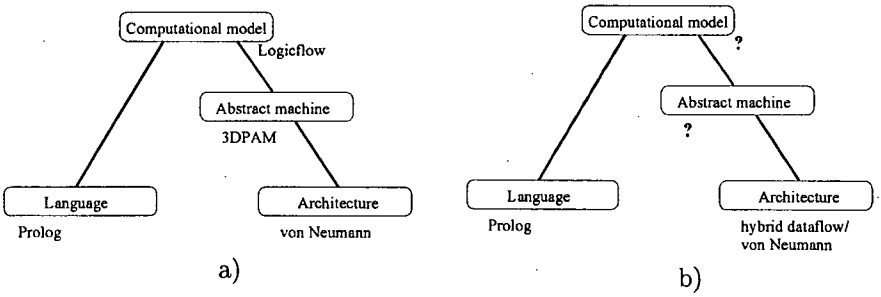


Figure 1: Levels of abstraction

However, the semantical gap between the models to be layers is still too big. The abstract machine, 3DPAM [14] provides the dataflow features required by the execution model at a high cost: token handling, queues, synchronisation, remote communication are realised by software.

Multithreaded architectures offer a solution for fundamental issues of distributed computing: eliminating idling at remote memory access and synchronisation [2]. An emerging class called hybrid dataflow/von Neumann tries to combine the speed of sequential execution and the simplicity and performance of dataflow scheduling [20]. The runtime model of hybrid dataflow/von Neumann architectures is close to that of Logicflow therefore, a natural step is making an attempt to replace the architecture to a hybrid one.

Hence the direction of engineering is reverse with respect to that of the LOGFLOW system: how the abstract engine can exploit the advantages of the architecture. Then how the execution model should be modified in order to fit the abstract engine making a real connection between the language and the architecture (Figure 1.b)?

The multithreaded and the hybrid properties of the architecture are completely independent. Multithreading enables remote memory accesses and thus, allows a new way of Prolog data layout. The main points of the new variable handling and some performance considerations have been presented in [16] and [17]. The hybrid property gives an opportunity for a new and efficient realisation of a Logicflow based model, where all the dataflow features are supported by the architecture. These features can be exploited at abstract machine level but accordingly, the Logicflow model must be be modified, too. The main steps of the modification in the abstract execution model has been presented in [18].

Yet, a set of very important questions remains open: how the original Logicflow and the modified Hybrid (Multithreaded) Logicflow models are related. Are they functionally equivalent? Does the Prolog Abstract Machine exactly what the execution model requires? Is the model sound? In this paper a part of the design is introduced in the framework of ASMs that shows how these issues can be handled and how the design process can be made precise and well documented by a proper formal method.

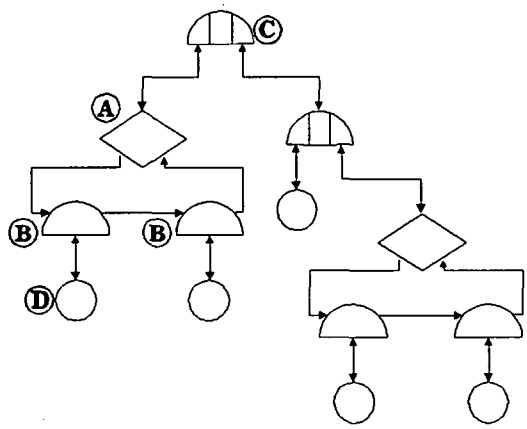


Figure 2: Elements of a DSG graph: Unify (A), And (B), Or (C) and Unit (D) nodes.

2.1 The Logicflow concept

The Logicflow model is a higher abstraction of dataflow principles [13] for a massively parallel (Or and pipeline-And) all-solution execution of Prolog programs on distributed memory architectures. Prolog programs are transformed into a Dataflow Search Graph (DSG, Figure 2). Nodes in this graph represent specific Prolog activities like unification, facts, handling alternatives, etc. Essentially they group together elementary dataflow nodes. As a consequence, DSG nodes can have inner state and one token is always enough to make a node fire.

In this model a clause is represented by a so called Unify-And ring. The Unify node (A in Figure 2) represents the head and the unification, And nodes (B) stand for the body goals and prepare the call. Alternative clauses are connected by Or nodes (C). The example graph in Figure 2 consists of 3 alternatives. Finally, group of consecutive facts are depicted by Unit nodes (D).

Logicflow is a Prolog model without backtrack. Request tokens (representing a query) are propagated from top to bottom. Or nodes duplicate the request tokens and thus, alternative branches of a predicate can be activated simultaneously. In this way Or-parallelism can be exploited. When request tokens reach the Unit nodes, they generate all the possible solutions to the request. Solution tokens form a stream flowing in the Unify-And ring. This ring can be considered as a pipeline: its different stages can process different tokens in the same stream in turn and thus, pipeline And-parallelism can be exploited as well. Solutions are propagated from bottom to top. Nodes must separate different token streams and manage their flow. Due to the all-solution property, there can be hundreds or thousands different token streams, each consisting of several tokens.

2.2 The hybrid dataflow/von Neumann Logicflow concept

The target architecture of the current Multithreaded Prolog Abstract Machine (MPAM) implementation is the Kyushu University Multimedia Processor on Datarol (KUMP/D) [24]. KUMP/D is a successor of multithreaded Datarol [1] and Datarol-II [15] machines. It is a hybrid dataflow / von Neumann one, i.e. it can support both program counter based sequential execution and dataflow scheduling. More precisely, the Datarol execution model distinguishes the short term and long term execution. Short term execution means sequential processing whereas long term execution is dataflow based scheduling of the sequential threads. In such a way there are threads that run exclusively until the termination point sequentially. At the end the next thread is scheduled on dataflow principles. In other words it is a kind of macro dataflow model, too.

In this model a program consists of simultaneously existing function instances. A function instance has its own context (frame) and shared code. Note, that the function instance and the thread are not the same: a function instance may consist of multiple threads. They belong to the same context. According to the definition of the thread, in a single context they do not work concurrently, rather the function can be considered as a set of consecutive threads. A thread is terminated whenever a synchronisation or remote memory access causes latency. At this point a fast context switch allows the processor to go on eliminating idle cycles; it is the essence of multithreading.

In the MPAM model each DSG node could be represented by a frame, where its own context is stored, furthermore it also has registers for token information. There is a thread (or more threads) attached to the frame. In such a way a node is represented as a function instance: the function code is realised by the threads whereas arguments and local variables of the function are kept in the frame.

A running function instance can activate (call) another function. It can pass arguments just like in case of procedure call of other programming models. When the new instance is ready to run, the scheduler may select it for execution. However, at this point, the caller instance remains as it is, its content is not stored in a token (in contrast with 3DPAM model). The new instance can proceed without load operations, because all the necessary data are available as arguments. A passive instance is waiting for some results. When the specified results are ready, it can be awaken on dataflow principles, where no load instructions are necessary: the state is the same as it was before, the results are local variables.

These are the principles of the MPAM working model that ensure an efficient interpretation of Prolog programs on the new architecture. However, the abstract execution model must be modified, too in order to narrow the semantical gap. There are three significant steps of redesign that enable the changes in the abstract machine model: creating node instances, optimised paths at alternatives and the introduction of aggregate nodes [18].

In Logicflow token streams form a central concept. Streams are maintained (and different streams are separated) by a colouring scheme and at abstract machine level the context tables that are needed for keeping consistent the colouring represent

some restrictions. However, the separation of streams could be defined in another way at abstract execution level. Obviously, if it could be guaranteed, that a node emits only one request token, there are no multiple reply streams and thus, they need not be separated. The key is in the Unify-And ring where And nodes prepare calls to predicates, i.e. they emit tokens towards Or, Unify or Unit nodes. If for each token in the stream a new instance of And node is created, the called node beneath it will receive a single request token. The Unify node merges the answer streams from the last And nodes within the ring. They belong, however, to the same stream representing the answer to the single request token of the Unify node. In such a way the token streams are separated physically without the need of colouring.

Or nodes (handling alternatives) do nothing at merging solution streams but maintain the correct colours. If the token colouring scheme can be eliminated, according to previous principles, there is no need to propagate solutions through the cascade of Or nodes, they can reach their root in one step.

As it was set forth the goal is to create an execution model, where nodes can be mapped to function instances easily. Although, it is possible at the present stage, increased granularity would reduce the cost related to instance (frame) management and data transfer between frames. The granularity can be increased by grouping together DSG nodes, resulting aggregate nodes. By a formal analysis 8 types of aggregate node have been defined as: unit, unify, or-unit, or-unify, and-or-unit, and- or-unify, and-unit, and-unify (Figure 3). In an aggregate node the component nodes share context and register information in a single frame saving significant time associated with frame set-up, communication, argument passing, intranode dataflow and so on. Thus, the unit of execution is an aggregate node that can be handled as a function instance with all the optimisations introduced before.

3 System design with ASMs

The idea of transforming the original Logicflow model has been presented in [18]. Yet, it is a rather informal description of the principles. The scope of current investigation is the verification of those transformation steps, in a broader sense, the question should be answered if the Logicflow and the Hybrid Multithreaded (HM) models of Prolog execution are semantically the same. Next, MPAM must be defined in such a way that it is equivalent to the HM Logicflow model. Abstract State Machines are proven to be capable, powerful and especially useful for solving this problem. They are able to deal with the very high level of abstraction of execution models and at the same time they are flexible enough to deal with MPAM at a significantly more concrete (with respect to implementation) level.

3.1 Abstract State Machines

Abstract State Machines represent a mathematically well founded framework for system design and analysis [3][6] introduced by Gurevich as evolving algebras [9].

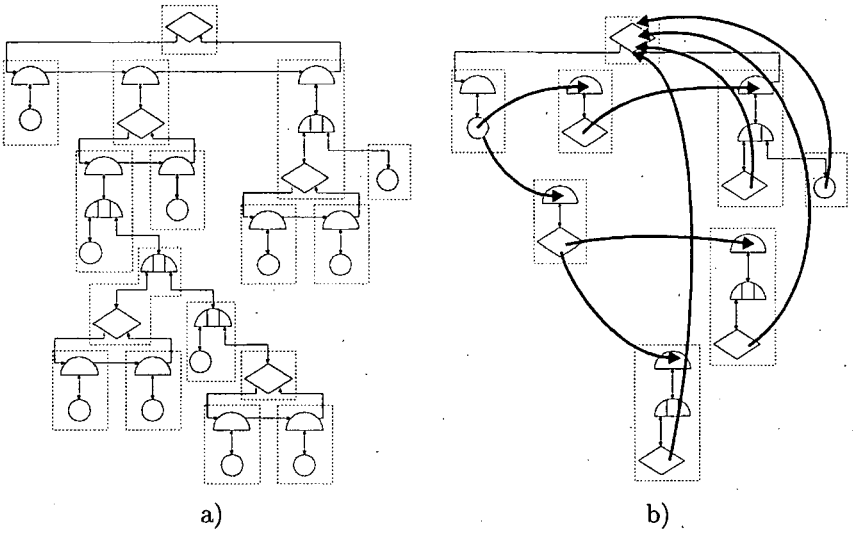


Figure 3: An example DSG graph where aggregate nodes are depicted as grey rectangles. The graph consists of 16 nodes instead of 30. All 8 types of aggregate nodes can be seen here (a). A snapshot of a possible execution showing node instances and optimised return paths (b).

The motivation for defining such a method is quite similar to that of Turing machines (TM). However, while TMs are aimed at formalising the notion of computable functions, ASMs are for the notion of (sequential) algorithms [12]. Furthermore, TMs can be considered as a fixed, extremely low level of abstraction essentially working on bits, whereas ASMs exhibit a great flexibility in supporting any degree of abstraction.

In every state based systems the computational procedure is realised by transitions among states. In contrast with other systems, an ASM state is not a single entity or a set of values but ASMs states are represented as (modified) logician's structures, i.e. basic sets (universes) with functions and relations interpreted on them. Experience showed that any kind of static mathematic reality can be represented as a first-order structure [12]. These structures are modified in ASM so that dynamics is added to them in a sense that they can be transformed.

Applying a step of ASM M to state (structure) A will produce another state A' on the same set of function names. If the function names and arities are fixed, the only way of transforming a structure is changing the value of some functions for some arguments. The transformation can depend on some condition. Therefore, the most general structure transformation (ASM rule) is a guarded destructive assignment to functions at given arguments [3].

ASMs are especially good at three levels of system design [3]. First, they help elaborating a ground model at an arbitrary level of abstraction that sufficiently rigorous yet easy to understand, defines the system features semantically and inde-

pendent of further design or implementation decisions. Then the ground model can be refined towards implementation, possibly through several intermediate models in a controlled way. Third, they help to separate system components. ASM is not a paper theory but it has been applied in various industrial and scientific projects like verification of Prolog [4] and Occam [5] compilers, Java virtual machine [23], PVM specification [7], ISO Prolog standardisation, validating various security and authentication protocols, VLSI circuits, and many more. The definition of ASMs is written in [8] and [11] and a tutorial can be found in [9]. A brief summary is presented here in order to make the paper self-contained.

A vocabulary (or signature) is a finite set of function names, each of fixed arity furthermore, the symbols *true*, *false*, *undef*, $=$, the usual Boolean operators and the unary function Bool. A state A of vocabulary Υ is a nonempty set X together with interpretations of function names in Υ on X. X is called the superuniverse of A. An r-ary function name is interpreted as a function from X^r to X, a basic function of A. A 0-ary function name is interpreted as an element of X.

In some situations the state can be viewed as a kind of memory. Some applications may require additional space during their run therefore, the *reserve* of a state is the (infinite) source where new elements can be imported inside the state.

A location of A (can be seen like the address of a memory cell) is a pair $l = (f, \mathbf{a})$, where f is a function name of arity r in vocabulary Υ and $\mathbf{a}$ is an r-tuple of elements of X. The element $f(\mathbf{a})$ is the content of location l.

An update is a pair $a = (l, b)$, where l is a location and b is an element of X. Firing a at state A means putting b into the location l while other locations remain intact. The resulting state is the sequel of A. It means that the interpretation of a function f at argument a has been modified resulting in a new state. This is how transition among states can be realised. An update set is simply a set of consistent updates that can be executed simultaneously.

ASMs are defined as a set of rules. The simplest rule is the skip that does not do anything. An update rule $f(a) := b$ is a rule and causes an update $((f, a), b)$, i.e. hence the interpretation of function f on argument a will result b. It must be emphasised that both a and b are evaluated in A.

A conditional rule R of form

```
if c then
   R1
else
   R2
endif
```

is a rule. To fire R the guard c must be examined first and whenever it is true R_1 otherwise, R_2 must be fired. A block of rules is a rule and can be fired simultaneously if they are mutually consistent.

An import rule of form

```
import v
   R
```

`endimport`

is a rule for introducing new elements from the *reserve* and firing rule R. The following construct

`extend` U `by` $v_1, ...v_n$ `with`
R
`endextend`

is a shorthand notation for

`import` $v_1, ...v_n$
$U(v_1) := true$
...
$U(v_n) := true$

R
`endimport`

that is new elements are imported from the *reserve* and they are assigned to universe U and then rule R is fired.

There are further rules introduced for convenience but these are the inevitable ones that will be used thoroughly in this paper. The basic sequential ASM model can be extended in various ways like nondeterministic sequential models with the choice construct, first-order guard expressions, one-agent parallel and multi-agent distributed models. The latter is applied in modeling Logicflow, therefore a very brief introduction follows.

A distributed ASM consists of

- a finite set of single-agent programs Π_n called modules
- a vocabulary Υ, which includes each $Fun(\Pi_n) - \{Self\}$, i.e. it contains all the function names of each module but not the nullary $Self$ function
- a collection of initial states

The nullary $Self$ function allows an agent to identify itself among other agents. It is interpreted differently by different agents (that is why it is not a member of the vocabulary.) An agent a interprets $Self$ as a while an other agent cannot interpret it as a. The $Self$ function cannot be the subject of updates.

A run of a distributed ASM is a partially ordered set M of moves x of a finite number of sequential ASM agents $A(x)$ which

- consists of moves made by various agents during the run. Each move has finitely many predecessors.
- The moves of any single agent are linearly ordered.
- Coherence: each initial segment X of M corresponds to state $\sigma(X)$ which for every maximal element $x \in X$ is obtainable by firing $A(x)$ in $\sigma(X - \{x\})$.

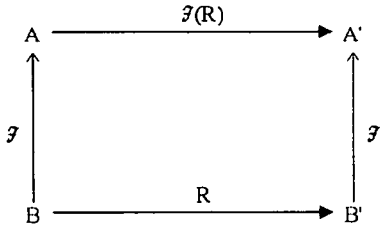


Figure 4: Principle of refinement.

3.2 Model refinement

Refinement is defined as a procedure where a "more abstract" and a "more concrete" ASMs are related according to the hierarchical system design. At higher levels of abstraction implementation details have less importance whereas they become dominant as the level of abstraction is lowered giving rise to practical issues. The goal is to find a controlled transition among design levels that can be expressed by a commuting diagram.

Let us assume ASM **M** has been refined to ASM **M'** by a partial abstraction function $\mathcal{F}$ that maps certain states of **M'** to **M** so that the diagram commutes in Figure 4. To put in another way, if the refinement is correct, it is the same if ASM **M'** moves from B to B' and then the corresponding state of ASM **M** is taken or first $\mathcal{F}(B)$ is taken and then the rule corresponding to R is fired. In both cases the result should be A'. However, the notion of equivalence, correctness and completeness strongly depends on the system designer's needs as it will be shown later.

4 From Logicflow to HM Logicflow model of execution

ASM represents the framework for proving the correctness of the new HM Logicflow model with respect to its predecessor Logicflow. Although the refinement procedure was introduced before as a transition between design levels, it is just a consequence of its "traditional" application. In fact, refinement is a method to relate any two ASMs of any level of abstraction. In our case Logicflow and HM Logicflow models that represent the same design levels, are related.

What has not been mentioned before is the considerable amount of intuition that is necessary at making a refinement step. Making a suitable mapping between corresponding states and rules is a hard task, if not impossible in case of complex systems. Instead, the gap between the two models should be divided by introducing submodels that differ only in one or two properties from the previous one and thus, a simple one-to-one mapping can be applied to some of the rules and states whereas the rest of mapping can be conceived by reasoning.

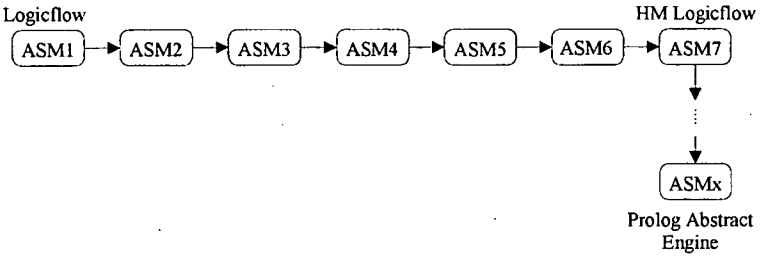


Figure 5: From Logicflow to HM Logicflow by a series of refinement steps.

As it was introduced earlier, the HM Logicflow model can be derived from Logicflow in three transformation steps. Yet, these steps from model to model are still to big because transformations involve the modification or replacement of many features at a time. Finally, a proper set of five submodels was found where the mapping can be done with reasonable efforts (Figure 5).

ASM1 is the original Logicflow model. In ASM2 a new kind of synchronisation is introduced in the Unify-And ring. As a consequence, in ASM3 instances of And nodes can be created. It yields that every node receives just one request token thus, there is no need for token colouring in ASM4. If there are no colours, the cascades of Or node can be optimised in ASM5. In ASM6 the concept of frames are brought into existence whereas ASM7 is the model for the HM Logicflow with all its details [19]. As it can be seen in Figure 5, all these models are at the same level of design abstraction.

The design of MPAM completely fits the same methodology. (It is entirely out of the scope of this paper and can be found in [19].) By successive refinement steps the abstract HM Logicflow model can be turned into a model of the engine that is much closer to the implementation level. The instructions of MPAM can be derived from groups of instructions of the ASM that describes it. It should be emphasised that in this scheme in Figure 5 both the model describing the principle of execution and the implementation details can be designed in the same formal framework of ASMs.

4.1 Description of the Logicflow model by an ASM

Logicflow is a distributed, dataflow and thus, indeterministic execution scheme that can be modeled as a distributed multi-agent ASM. There are two questions to be clarified:

1. From which point of view should the model be described, i.e. what entities should be the agents? The system could be modeled as the graph nodes are agents and react to the incoming tokens. Another possibility, that was chosen finally, is where tokens are agents and they make the nodes fire. Although this issue must be clearly answered before building the model, the decision is rather the question of taste.

2. What should the ASM describe? In [13] the function of each DSG node is described and it is claimed that every Prolog program can be compiled to a set of such nodes. Yet it is not a description of the Prolog execution. For instance the DSG graph for a Prolog program that contains recursion (and most Prolog programs do) may be different for different input parameters although the DSG components (the compiled program) and their functionality are the same. The ASM model aimed at simulating the actual execution, therefore it describes how the certain DSG graph is constructed from the precompiled building blocks.

The machine created for modeling Logicflow is called ASM1. There is just one module thus, each agent executes the same program. Furthermore, the number of agents changes during the execution as tokens are created and discarded. The $Self$ function is realised by the nullary function t, i.e. it means the current token that realises the agent and the same t in the program text is interpreted differently for different agents.

4.1.1 The basic sets and functions

ASM1 consist of the following universes:

- $TOKEN$. Elements in this set are the agents. The nullary function t represents the $Self$ function. Tokens have type and colour. The unary function $type : TOKEN \rightarrow \{DO, SUB, SUCC, FAIL, FAIL2\}$ [1] and $colour : TOKEN \rightarrow COLOUR$ can retrieve the type and colour of the given token, respectively. $loc : TOKEN \rightarrow NODE$ returns the current location (node) of the token. It is assumed that tokens are always assigned to a node and there is no buffering or transition time between two nodes. Some tokens can carry environments, i.e. variable substitutions that can be obtained by the subst: $TOKEN \rightarrow SUBSTITUTION$ function.

- $NODE$. This universe contains the nodes that realise the actual DSG graph. There are 5 types of them that can be retrieved by $node : NODE \rightarrow \{AND, OR, UNIT, UNIFY, QUERY\}$. $mode : NODE \rightarrow \{create, active\}$ is related to the construction of the DSG graph and results if the graph connected to the node has been built already or not. The function $returnport : NODE \rightarrow \{reply.in, reply.in1, reply.in2\}$ gives the port type where the actual node must return the answer tokens. The topology of the nodes can be described by the $on\_arc : NODE \times INT \rightarrow NODE$ function and by the macros derived from it:

  - $child(node) \equiv on\_arc(node, 3)$
  - $child1(node) \equiv on\_arc(node, 3)$

[1] FAIL and FAIL2 tokens are functionally equivalent and they are not distinguished in [13]. The introduction of FAIL2 tokens is simply a notation for making the explanation easier. FAIL2 tokens are those occuring in a Unify-And ring.

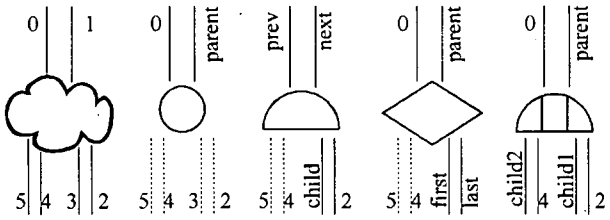


Figure 6: A generic node and interpretations of arc labels for different types of nodes.

- $child2(node) \equiv on\_arc(node, 5)$
- $prev(node) \equiv on\_arc(node, 0)$
- $next(node) \equiv on\_arc(node, 1)$
- $parent(node) \equiv on\_arc(node, 1)$
- $first(node) \equiv on\_arc(node, 3)$
- $last(node) \equiv on\_arc(node, 2)$

This kind of description assumes a generic node with 6 arcs as it can be seen in Figure 6. The actual types of nodes enumerate their arcs accordingly, though not all arcs are in use. In such a way a kind of navigation can be defined among nodes, their relationship (parent-child, previous- next) can be described precisely yet, in a readable form.

Some nodes contain context information like colour, substitution, counter that can be retrieved by the appropriate functions ($colour\_context : NODE \times COLOUR \rightarrow COLOUR$, $subst\_context : NODE \times COLOUR \rightarrow SUBSTITUTION$, $counter : NODE \times COLOUR \rightarrow INT$, $and\_state : NODE \times COLOUR \rightarrow \{open, closed\}$, $or\_state : NODE \times COLOUR \rightarrow \{wait1, wait2\}$).

- *COLOUR*. Token streams are separated by a colouring scheme. Tokens forming a stream have the same colour no matter what the actual type or content of the token is.

- *STREAM*. Tokens of the same colour targeted to the same port of a node form a stream. It is essentially a set. Tokens in this set can fire the node they are waiting for in arbitrary order except that a *FAIL* token must be the last one terminating the stream. A stream can be identified by the node and port the tokens are waiting for and the colour information ($stream : NODE \times PORT \times COLOUR \rightarrow STREAM$). The relation $instream : STREAM \times TOKEN \rightarrow \{true, false\}$ is true if the given token is a member of the stream, whereas function $card : STREAM \rightarrow INT$ returns the number of tokens in the stream.

- $SUBSTITUTION$. Substitution is a set of variables and their binding values.

- $PORT$. A port is an entry point to a node from the following set: $PORT = \{request.in, reply.in, reply.in1, reply.in2\}$.

- $LIT, CLAUSE$. Literals and list of literals, i.e. clauses. Function $procdef : LIT \rightarrow CLAUSE^*$ returns the definition for the given literal. A clause can be separated to head and body parts by the $head : CLAUSE \rightarrow LIT$ and $body : CLAUSE \rightarrow LIT^*$ functions. There is a predicate or goal assigned to some nodes that can be retrieved by $predicate : NODE \rightarrow CLAUSE^*$ and $goal : NODE \rightarrow LIT$, respectively.

4.1.2 Modeling Logicflow by ASM1: an example

A simple example program is presented here step-by step that shows the most important features of the ASM1 model. Abstract State Machines can be treated as a kind of pseudo-code so, even if one is not familiar with all the details of ASMs, the code can be read easily and it is self-explanatory more or less (see Appendix A.)

The example given here is the well-known family program:

```
grandfather(X,Y):-father(X,Z),parent(Z,Y).
parent(A,B):-mother(A,B).
parent(A,B):-father(A,B).
father(bill, john).
father(bill, james).
father(john, jack).
mother(jane, jack).
mother(alice, fred).
mother(jane, charles).
:-grandfather(X, jack).
```

The execution starts with one Do token (agent) at the Query node (Figure 7.a). There are no other tokens or nodes in the system. The activator of the Do token is `grandfather(X, jack)`, i.e. the query, and the substitution is empty. This initial state is represented by the following structure:

$type(loc(t)) = Query$
$type(t) = DO$
$subst(t) = \{\}$
$act(t) = grandfather(X, jack)$

In this case rule 14 can fire extending the graph with a new but untyped node. This operation is realised by the extend construct that brings a new element from the reserve (and this element is different from those already in some basic set) and puts it into a set, $NODE$ in this example. The relationship between the Query

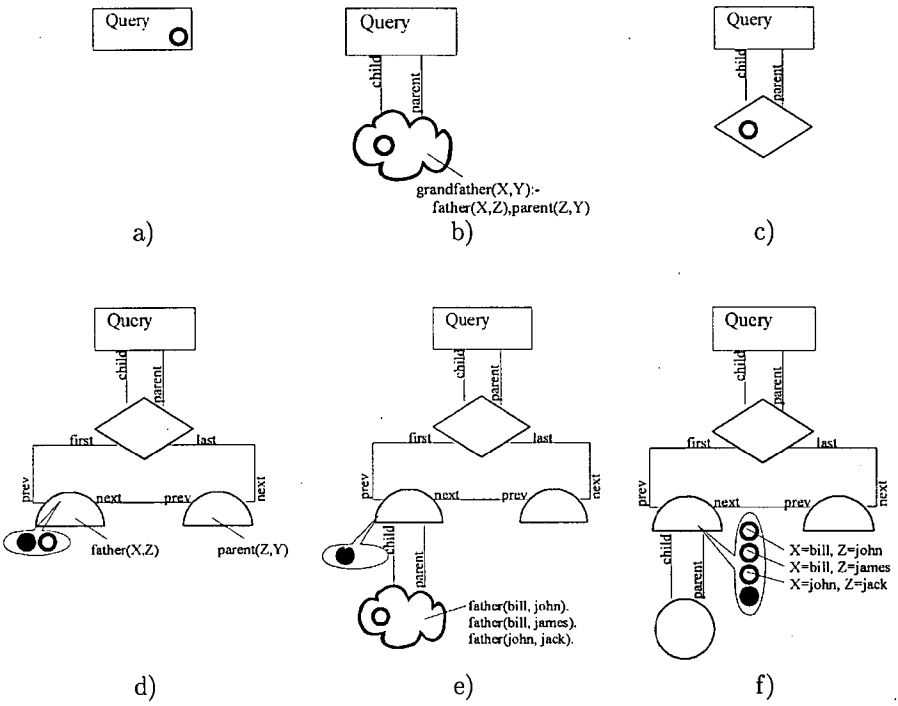


Figure 7: The initial state and states after firing rules 14, 13, 7a, 2a, 13, 3, 1 respectively. The result of rules 13, 3, 1 is shown together in f).

already in play and the new node is set by the macro variations of the *on_arc* function. The predicate assigned to this node is `grandfather(X,Y):-father(X,Z), parent(Z,Y)`and the Do token is moved to it (Figure 7.b). Then rule 13 can fire that sets the type of current undefined node to Unify (the predicate is a single clause having body.) This change enables rule 7a to fire. Note, the Unify node is in *create* mode, i.e. it is the first time a token appears on it and the subgraph must be extended. The Unify node represents the head of a clause where unification takes place. If the unification of the activator of the token is successful with the head of predicate (and it is in this case), the graph is extended by And nodes resulting the Unify-And ring. Each And node in this ring is in *create* mode, their connecting arcs are set and body goals are assigned to them. The current colour and the substitution (updated with θ, the most general unifier) of the token are saved and a new colour is assigned to it. Note that the new colour is obtained by the `extend` construct which guarantees that this colour is different from all previous ones. A stream is created towards the *request.in* port of the first And node in the ring and the current token (transformed into a Sub type, substitution is θ and location is the And node) is put into it together with a terminating Fail2 token (Figure 7.d).

At this point rule 2a can fire. The node sets its counter to 1 (the number

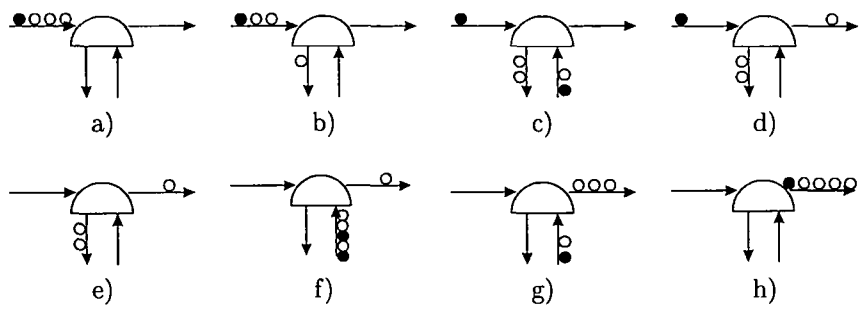


Figure 8: Working cycle of an And node in Logicflow model

of received Sub tokens) and since it is in *create* mode (see the extension in rule 7a) it produces its subgraph. The subgraph is untyped currently, and the assigned predicate is `father(bill, john), father(bill, james), father(john, jack)` (Figure 7.e).

Hence two rules may fire simultaneously. Rule 13 sets the type of untyped node to Unit (a predicate with multiple clauses and none of them have a body), whereas rule 3 sets the state of the And node closed and the Fail2 token vanishes. As a consequence of rule 13, rule 1 can fire. A Unit node brings together the successive facts in the program and produces all the possible solutions to them. It creates a stream to its parent node (i.e. the And node) and puts the solution tokens (Succ tokens) into it. Each Succ token has the same colour and they are identical with exception for the different substitutions according to the result of three different unifications. Finally, a Fail token is put into the stream terminating the computation (Figure 7.f).

The first steps of executing the example program showed the most important features of the ASM1 model that describes the Logicflow model. The reader may trace the execution further by applying the appropriate rules in Appendix A.

4.2 The first submodel: ASM2

The first model that has been introduced between the Logicflow and the Hybrid Multithreaded Logicflow introduces a different way of synchronisation in the Unify-And ring. The basic philosophy of Logicflow is that for a token representing a goal an answer *stream* of the same colour, terminated with a Fail token, is expected at the reply arc of the node no matter if it was produced by a single node or by a large subgraph. The receipt of the Fail token means that all the possible solutions to the query has been found and there are no active tokens belonging to the computation in the subgraph.

Ensuring this property in the Unify-And ring is a complex task. There can be multiple overlapping streams in the ring separated and identified by colours. An And node receives a token stream and must generate a token stream of the same colour making it sure that the Fail2 token appears on its reply arc only when

no more solutions are possible. This is realised by *and_state* and a counter. The counter contains the number of tokens sent to the subgraph, i.e. each incoming request token increments it (Figure 8a, b, c) whereas each terminating Fail token in the answer stream decreases it (Figure 8 d, g, h). The Fail2 token on the request arc of the And node makes its state closed meaning that no more request tokens can be expected (Figure 8 e). At this point whenever the counter is 0, the Fail2 token can be sent on the reply arc terminating the answer stream (Figure 8 h).

This solution can be considered as a distributed tracking of the active streams in the Unify-And ring. A single Fail2 token is circulated in the ring terminating the request/reply stream and whenever it hits the Unify node, there are no more tokens belonging to the same task in the subgraph. (This property has been proven in [13].)

However, if And node *instances* are created for each token in the stream according to the modifications, this mechanism is not viable, since there is no single route for tokens in the ring and thus, there cannot be a single termination signal at the end. The first step in the modifications is the redesign of the synchronisation mechanism in the Unify-And ring.

States and counters in the And nodes, furthermore Fail2 tokens are not necessary anymore. Instead, a counter is introduced in the Unify node that keeps a record of the active streams in the ring. A new type of nodes is introduced as Last_And which is the last and node within the Unify-And ring. Each time an And node receives a solution token, it increments the counter in the Unify node. Each time an And or Last_And node receive a Fail token, they decrement the counter. The functionality of And and Last_And nodes is equivalent except that Last_And nodes never increment the counter.

ASM2 is the model that describes the Logicflow model where this slight modification is introduced. Most rules remain intact except those related to And or Unify nodes. (See Appendix B.)

The modification of the synchronisation mechanism within a Unify-And ring seems to be simple, feasible and correct. But can it be shown formally that ASM1 and ASM2 are equivalent and functionally they do exactly the same?

5 Proof of equivalence of ASM1 and ASM2

This proof represents the first element in the series of equivalence proofs in Figure 5. It is introduced here as a kind of case study and further proofs can be carried out in a similar way. First, it should be clarified what equivalence means. Then it must be defined how the indeterministic behaviour of these models can be treated. While the latter issue is general in the whole proof procedure, the first one is unique for each step, i.e. two model can be said equivalent with respect to some definition. Obviously, these definitions involve the property that has been changed in the given refinement step.

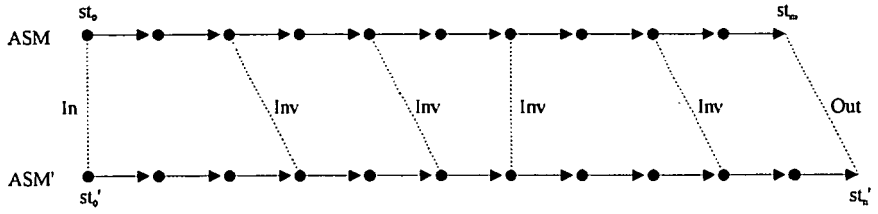


Figure 9: Schellhorn's modularisation theorem

5.1 The notion of equivalence

There can be many definitions of equivalence according to the level of abstraction and it is even possible that two algorithms are identical to some definitions of equivalence and different to others [10]. ASMs offer a possibility to precisely define what equivalence means in the given situations. One presumption for the equivalence is that the two algorithms produce the same output for the same input. In [10] there are two possible equivalencies defined, the strict lock-step equivalence and the lock-step equivalence. It is shown that the two algorithms in scope (variations of bounded buffers) are lock-step equivalent but not strict lock-step equivalent. In both cases every step of algorithms and the corresponding states are taken into consideration which is not feasible for real life complex applications.

A more practical approach that can be applied at refinements is presented in [21]. Let us assume two relations IN and OUT of initial and final states, respectively. A refinement is correct if for every finite trace of $ASM'(st'_0, ...st'_n)$ and for every st_0 of ASM with $IN(st_0, st'_0)$ there exists a finite trace of ASM $(st_0, ...st_m)$ so that $OUT(st_m, st'_n)$. In other words: let us take into consideration all valid runs of ASM' starting from st'_0 and ending in st'_n. For each such run let us take all the st_0 states of ASM that are in relation $IN(st_0, st'_0)$. If every run of ASM starting from st_0 ends in state st_m that is in relation $OUT(st'_n, st_m)$ then the refinement of ASM to ASM' is correct. If the refinement of ASM' to ASM is correct, too, then it is complete. Although, it is just a correctness of a kind of model transformation, it is also a definition of equivalence based on input-output behaviour.

Schellhorn's main invention is the generalised proof method for refinement correctness (the "Modularisation Theorem"). The commuting diagram can be partitioned by finding states that are in arbitrary relation which is the so called coupling invariant (Figure 9). In such a way the correspondence between two computations can be reduced to subcomputations, i.e. if the two ASMs are started from related states they should finish their computation in related states as well. Schellhorn formalised his theorem for deterministic and indeterministic ASMs and defined the trace correctness as well. In the followings Schellhorn's idea is applied for equivalence proof.

5.2 The problem of indeterminacy

Logicflow model (and its modified versions) is inherently indeterministic due to dataflow nature. The corresponding ASM models are distributed multi-agent ones with similar behaviour. A program is deterministic if for some input set it generates the same output set no matter how many times the program is executed. Yet, the execution still can be indeterministic reaching the output set in different ways (different order of state transitions) from run to run. The main problem is that some execution paths can lead to the correct output set while others do not.

Schellhorn's theorem for deterministic ASMs says that if there are two states x and x' that are in relation by the coupling invariant, there exist two integers i and j so that after i step of **ASM** and j step of **ASM'** the coupling invariant holds for the resulted states in order to clain the refinement correct. However, it is just one possible successor state. Shellhorn generalizes his theorem for indeterministic behaviour so that for every possible x'_j (the resulted state after j steps) there must be an i so that x'_j and x_i are in relation.

What does it mean? It is not enough to find one possible partition of the commuting diagram but all the possible partitions. A serialisation method will be used according to [3]: given any initial segment of a run, each linearisation has the same final state. It is a consequence of the coherence condition in the definition of distributed ASMs. The execution is serialised, and then the partitions can be obtained according to the principles of partitioning deterministic systems. In this case no special properties of the actual serialisation can be used, because it is not *one* linearisation but *any* of them. In other words the partitioning will yield all the subdiagrams of which all the possible linearised executions of the two ASMs can be constructed.

5.3 Definition of equivalence of ASM1 and ASM2

Obviously, two Prolog executions are equivalent if they produce the same solutions to a given query. (Due to the all-solution property of the Prolog models in scope and the absence of side-effects the order in which solutions are given is meaningless.) However, taken into consideration two facts, several rules can be omitted at the proof thus significantly reducing the size of the commuting diagram.

First, there are several rules that are identical in ASM1 and ASM2. It is the consequence of careful insertion of submodels where special attention was paid for introducing small changes from model to model. Evidently, they do not affect the equivalence of the two ASMs. Furthermore, ASM rules are local in a sense that they modify the state of the current token and the node it is currently on and do not affect other tokens or nodes in any way.

As a consequence, the equivalence of the two models can be proven by showing the equivalence of the working cycle of Unify-And rings. It can be assumed that the embedding graph behaves the same in the two cases and there are no interactions among different Unify-And rings. First, let us assume that there are no other Unify-And rings in the subgraph attached to the Unify-And ring in question. If they are

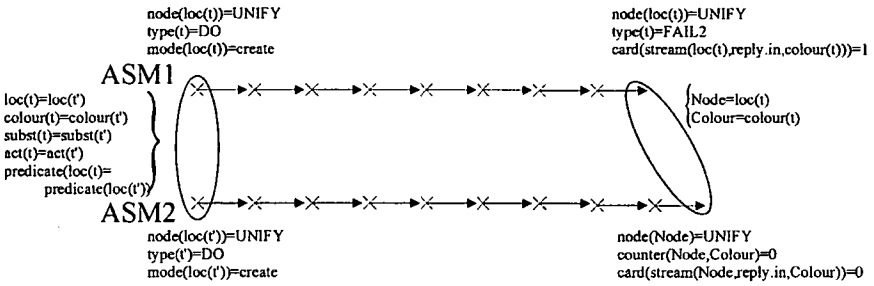


Figure 10: The reduced commuting diagram.

proven to be equivalent, they behave exactly the same as a Unit node with respect to the generation of a token stream and then the equivalence holds in case of any embedding graph.

ASM1 and ASM2 are equivalent if and only if for each valid run of ASM1 there is a related run of ASM2. According to the reasoning above, this definition can be narrowed as: the two models are equivalent if and only if for each valid run of a Unify-And ring in ASM1 there is a corresponding run of the same Unify-And ring in ASM2.

In both cases the initial state is represented by the appearance of a Do token at the Unify node. The final state in ASM1 is the appearance of a Fail2 token on the reply.in arc of the Unify node, whereas the related state in ASM2 is composed by the 0 state of the counter and the empty stream. (Note that in ASM2 there is no Fail2 token on the Unify node, that is why the condition is expressed as a first-order formula in rule 24.) The correspondence is expressed by the same properties of tokens and nodes as it can be seen in Figure 10.

5.4 Partitioning the commuting diagram

In Figure 10 a reduced commuting diagram can be seen. It is reduced in a sense that state transitions occurring in the Unify-And ring are included only. It shows one possible sequence of execution and as it has been explained, the proof should cover all possible execution patterns.

The most important and generally the most difficult task is finding the proper coupling invariant. It can be any property that relates a state of ASM1 to ASM2 and by which the diagram can be partitioned in such a way that from the initial state a related pair of states can be reached in finite steps, then the invariant property holds for some pairs of states, finally, from a related pair of states the final relation can be reached (see Figure 9). The essential change in the first submodel is the introduction of a single (centralised) counter in the Unify node instead of the many (distributed) counters and state flags of And nodes. Therefore the coupling invariant should be associated to the semantics of the counter. The centralised counter maintains the number of active streams in the ring which is essentially the sum of distributed counters in And nodes.

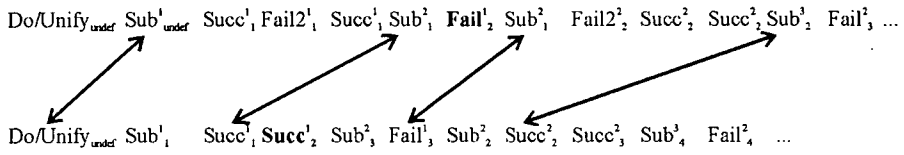


Figure 11: Initial fragment of the commuting diagram for the example runs

Let s_1 the sum of counters in And nodes belonging to the same ring and to the same colour in ASM1 and let s_2 the current value of counter in Unify node of the same colour in ASM2. Let X an initial segment of a run of ASM1 with maximal element m and Y an initial segment of a run of ASM2 with maximal element n. Then $s_1(X)$ means the value of s_1 after performing the steps of X, the meaning of $s_2(Y)$ is similar.

The coupling invariant relates states belonging to initial segments $X - \{m\}$, $Y - \{n\}$ where the value of the counter changes so that

$s_1(X - \{m\}) \neq s_1(X)$, $s_2(Y - \{n\}) \neq s_2(Y)$

and after these steps they are equal

$s_1(X) = s_2(Y)$

then $\sigma(X - \{m\})$ and $\sigma(Y - \{n\})$ are in relation where σ is a projection from segments (sequences of steps) to states.

Let us introduce the following notation: $Token^n_x$ is an event when token of type Token is received at the nth And node in the Unify-And ring with s_1 or $s_2 = x$, whereas Token/Unify means an event caused by a token of type Token at the Unify node. ASM rules are usually guarded by dataflow firing conditions, therefore for the sake of simplicity these events will represent firing the rules. A possible valid run of ASM1 in the Unify-And ring with 3 And nodes is the following list of events:

$Do/Unify_{undef} - Sub^1_{undef} - Succ^1_1 - Fail2^1_1 - Succ^1_1 - Sub^2_1 - Fail^1_2 - Sub^2_1 - Fail2^2_2 - Succ^2_2 - Succ^2_2 - Sub^3_2 - Fail^2_3 - Succ^2_2 - Succ^3_2 - Sub^3_2 - Sub/Unify_3 - Sub^3_3 - Fail^2_4 - Fail2^3_3 - Fail^3_3 - Succ^3_2 - Succ^3_2 - Sub/Unify_2 - Fail^3_2 - Sub/Unify_1 - Succ^3_1 - Fail^3_1 - Sub/Unify_0 - Fail2/Unify_0$

A corresponding run in ASM2 can be obtained by omitting Fail2 events. It means that the embedding graph received and produced tokens in exactly the same timing.

$Do/Unify_{undef} - Sub^1_1 - Succ^1_1 - Succ^1_2 - Sub^2_3 - Fail^1_3 - Sub^2_2 - Succ^2_2 - Succ^2_3 - Sub^3_4 - Fail^2_4 - Succ^2_3 - Succ^3_4 - Sub^3_4 - Sub/Unify_4 - Sub^3_4 - Fail^2_4 - Fail^3_3 - Succ^3_2 - Succ^3_2 - Sub/Unify_2 - Fail^3_2 - Sub/Unify_1 - Succ^3_1 - Fail^3_1 - Sub/Unify_0 - 0/Unify_0$,

where 0/Unify means rule 24.

If both ASM1 and ASM2 were deterministic models, the commuting diagram that is represented in Figure 11 could be easily partitioned by the invariant property showing the equivalence of the two models. But due to the indeterministic nature, the partitioning should be possible for every linearisation of valid runs of ASM1 and ASM2. Therefore, from these strings the general properties must be extracted

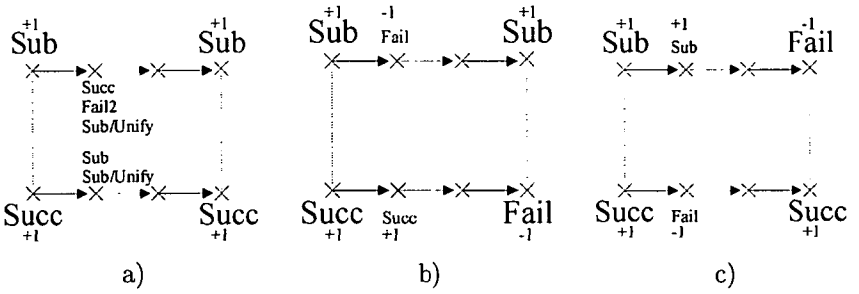


Figure 12: Straight (a) and inverted (b) pairs of states. Inverted pairs (c) is impossible.

omitting the special features of the certain runs. In such a way a general partitioning will be resulted that can be treated as "building blocks" from which all the possible runs can be constructed. If the commutativity for these subdiagrams holds, it holds for all the diagrams that can be constructed of them. (Note that all possible runs can be constructed but not every possible construction is a valid run.)

First it must be shown that from the initial states a related pair of states can be reached. According to the definition, the first Sub event of ASM1 will be related to the Do/Unify event of ASM2 (Figure 13.a.) Hence there are two possibilities: the subgraph connected to the first And node can produce a solution (Figure 13.b) or not (Figure 13.c). Hence at the beginning either a+b or a+c subdiagrams are fixed sequences. The terminating subdiagram d is the only possible terminating sequence and it needs some explanation.

Between the last Fail and the terminating states there cannot be anything except Sub/Unify and Fail2 events. Otherwise, if there were any Sub or Succ events, they would be preceded by their terminating Fail event that is impossible. In such a way the only terminating sequence is subdiagram d in Figure 13. It also shows that the final state can be reached from a related pair of states (namely a Fail-Fail pair).

In ASM1 the relevant states where the counters are modified in any way are represented by Sub and Fail events. Between them any number (including 0) of irrelevant Succ, Fail2 and Sub/Unify event can occur without affecting the sum of counters. In case of ASM2 those events are Succ (except at the last And node in the ring) and Fail with any number of Sub and Sub/Unify (and Succ at the last And node) between them.

In the straight case there are no relevant states between related states (Figure 12.a). It means that at the related states both s_1 and s_2 are incremented or decremented. Therefore, subdiagrams e, f, h, i in Figure 13 can be easily and systematically created. However, it is possible that states are inverted, there are relevant states that are not related, i.e. the next relevant state increments the counter in an ASM and decrements in the other (Figure 12.b, c). It is even possible that there are multiple inverted states. An example for inversion can be seen in bold in Figure 11.

What does it make evident that after an inversion a related pair of states will be reached in Figure 12.b?

Remark 1. Let $\sigma(X)$ and $\sigma(Y)$ be related states. The number of Fail events in X and Y are equal. (Proof: starting from subdiagram a) in Figure 13 and applying straight subdiagrams, the statement holds. Reaching the first inverted pair, the appearance of a Fail event in one ASM guarantees the existence of another Fail event in the other ASM model, because the number of Fail events in the entire runs are equal.)

Remark 2. Succ events are in relation with Sub events that happened later in the run. It is simply a consequence of the fact that Sub events are caused by Succ events (except the first one.)

From these two statements it is true that for the Fail event of ASM1 in Figure 12.b there is another Fail event in ASM2 and for Succ in ASM2 there must be a Sub in ASM1. In such a way the relation holds for the Sub-Fail pair.

On the other hand, the another combination of inverted states is not possible (Figure 12.c). The second Sub event in ASM1 would have been related (through an inversion) to a Succ event that occurs later which is in contradiction with the causality expressed in Remark 2.

As it can be seen, both the start-up and the final stage are of given, fixed types of subdiagrams. There are 3 types of related pairs: Sub-Succ, Fail-Fail and Sub-Fail. From these 9 other types of subdiagrams can be created. In such a way all the valid runs can be constructed from these 13 types of subdiagrams.

For a single diagram, e.g. e) in Figure 13 Schellhorn's theorem states the following. Starting ASM1 from Sub and ASM2 from a related Succ, for every possible successor state in ASM1 there must be a successor state in ASM2 so that the invariant holds again. This is covered by diagrams e), h) and k) in Figure 13. There can be any number of intermediate states, reaching the next Sub/Succ, Fail/Fail or Sub/Fail pair, the relation is true. Hence the proof can be continued starting from the new related states. It is easy to trace the correctness from the very beginning to the end. The commutativity of subdiagrams shows the commutativity of all diagrams constructed of them that means the equivalence of ASM1 and ASM2 according to our definition.

In this step it was assumed that there are no other Unify-And rings in the subgraphs connected to the And nodes in scope. Since it was shown that the Unify-And ring of the modified model behaves like the one in the original Logicflow and hence, from the parent node's point of view there is no difference between a Unit and a Unify node, the equivalence proven here is true even if there are Unify-And rings in the subgraphs attached to the And nodes.

6 Conclusion

In this paper a small part of the design of a distributed parallel Prolog execution model was introduced where Abstract State Machines were applied in the course of development. The outcome of the paper is not a description of a parallel model

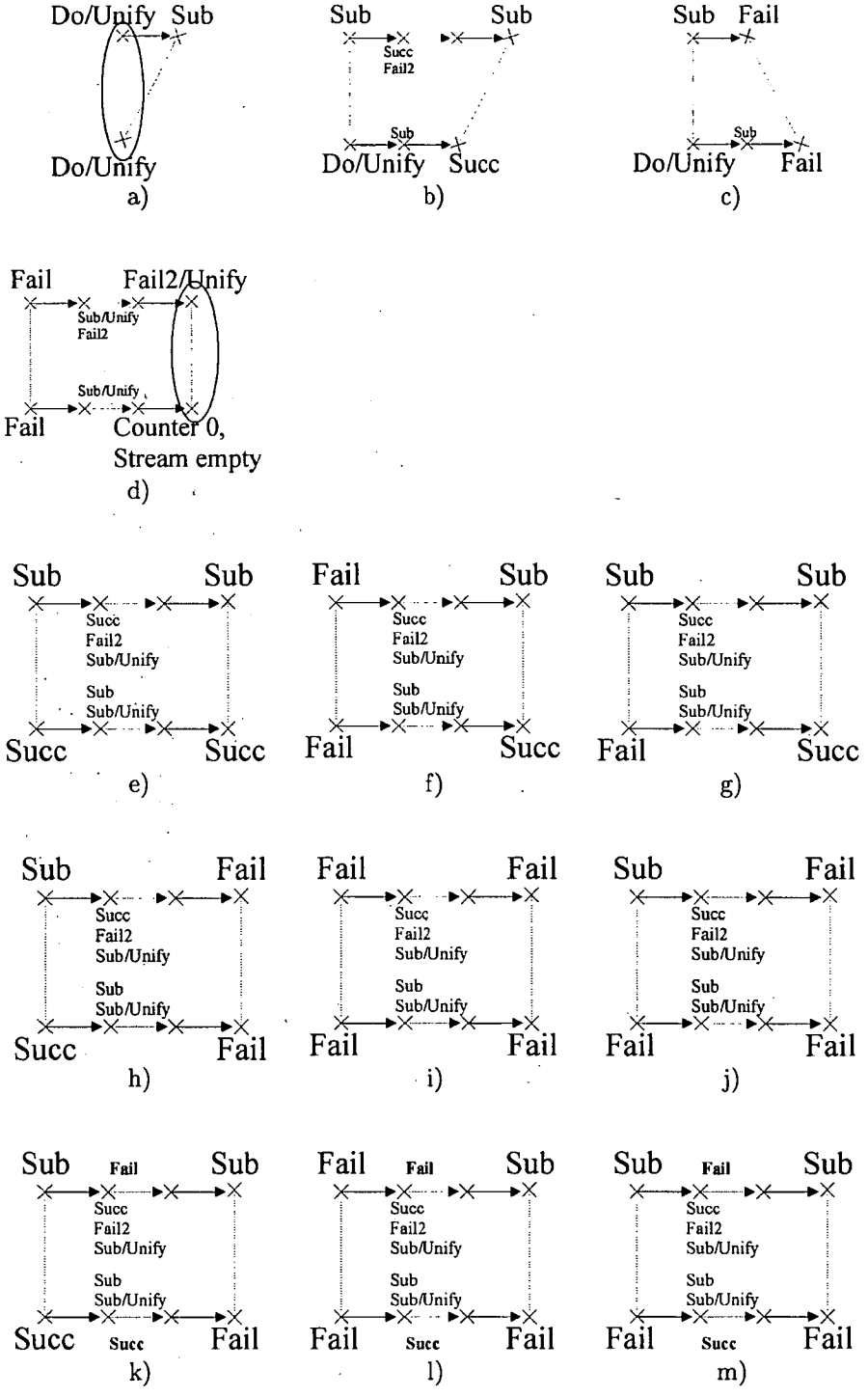


Figure 13: Commuting subdiagrams

ready to be implemented, rather a profitable case study where the application of ASM is demonstrated in different situations.

First, an existing model, Logicflow has been described by ASM1. It is a precise and succinct way of specification that helps to discover the features of the system. Then, this model can be derived to another one by successive minor modifications that are realised by a series of submodels represented by ASMs. In this paper a single step of such modification was introduced. The ASM notation makes clear the scope and the extension of changes.

ASMs are not just a method for description and analysis but provide a framework where models can be compared and their equivalence or inequivalence can be precisely defined and proven. In the current context the equivalence of ASM1 and ASM2 has been proven. The proof method was able to tackle with the distributed and indeterministic nature of dataflow based parallel Prolog models. In summary, experience showed the endowment and efficiency of ASMs (and the related techniques) in system design.

Acknowledgements

Author would gratefully express his appreciate to Prof. Egon Börger who helped with the very first steps in ASMs and gave valuable suggestions and inspiration for the entire work. Prof. Ferenc Vajda and Gábor Dózsa have always been ready for discussion and helped with presenting the work. Earlier stages of the multithreaded Prolog system were investigated in a joint project with Kyushu University where Prof. Makoto Amamiya and Hiroshi Tomiyasu were especially helpful.

References

[1] M.Amamiya, R. Taniguchi: Datarol: A Massively Parallel Architecture for Functional Language. Proc. Second IEEE Symposium on Parallel and Distributed Processing, 1990, pp. 726-735.

[2] Arvind and R.A. Ianucci: Two fundamental issues in multiprocessing. Proc. DFVLR Conf. on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, 1987. pp. 61-88.

[3] E. Börger: High Level System Design and Analysis using Abstract State Machines. In: D. Hutter et al. eds., Current Trends in Applied Formal Methods (FM-Trends 98), LNCS 1641, Springer, pp. 1-43.

[4] E. Börger, D. Rosenzweig: The WAM - Definition and Compiler Correctness. In: C. Beierle and L. Plümer eds, Logic Programming: Formal Methods and Practical Applications, Studies in Computer Science and Artificial Intelligence, chapter 2, North Holland, 1994, pp. 20-90.

[5] E. Börger, I. Durdanovic: Correctness of Compiling Occam to Transputer code. Computer Journal, Vol. 39, No. 1, Oxford University Press, 1996, pp. 52-92.

[6] E. Börger: Why Use Evolving Algebras for Hardware and Software Engineering? In M. Bartosek, J. Staudek, J. Wiederman, editors, Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics, LNCS 1012, Springer, 1995, pp. 236-271.

[7] E. Börger and U. Glässer: Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras. In: Y. Gurevich and E. Börger, Evolving Algebras Mini-Course, Technical Report BRICS-NS-95-4, BRICS, University of Aarhus, July 1995. http://www.eecs.umich.edu/gasm/papers/pvm.html

[8] Y. Gurevich: Evolving Algebras 1993: Lipari Guide. In: E. Börger ed., Specification and Valdation Methods, Oxford University Press, 1995. pp. 9-36.

[9] Y. Gurevich: Evolving Algebras: An Attempt to Discover Semantics. In: G. Rozenberg, A. Salomaa eds., Current Trends in Theoretical Computer Science, World Scientific, 1993, pp. 266-292.

[10] Y. Gurevich, J.K. Huggins: Equivalence is in the Eye of Beholder. Theoretical Computer Science, Vol. 179, No. 1-2, Elsevier, 1997, pp. 353-380.

[11] Y. Gurevich: May 1997 Draft of the ASM Guide. http://www.eecs.umich.edu/gasm/papers/guide97.html

[12] Y. Gurevich: Sequential Abstract State Machines Capture Sequential Algorithms. ACM Transactions on Computational Logic, Vol. 1, No. 1, July 2000, pp. 77-111.

[13] P. Kacsuk: Execution Models for a Massively Parallel Prolog Implementation. Journal of Computers and Artificial Intelligence, Vol. 17, No. 4, Slovak Academy of Sciences, 1998, pp. 337-364 (part 1) and Vol. 18, No. 2, 1999, pp. 113-138 (part 2)

[14] P. Kacsuk: Distributed Data Driven Prolog Abstract Machine. In: P. Kacsuk, M.J.Wise eds., Implementations of Distributed Prolog. Wiley, 1992. pp. 89-118.

[15] T. Kawano, S. Kusakabe, R. Taniguchi, M. Amamiya: Fine-grain multithread processor architecture for massively parallel processing. Proc. First IEEE Symp. High Performance Computer Architecture (HPCA95), Raleigh, 1995, pp. 308-317.

[16] Zs. Németh, P. Kacsuk: Analysis and Improvement of the Variable Binding Scheme in LOGFLOW. In: I. de Castro Dutra, M. Carro, V. Santos Costa, G. Gupta, E. Pontelli, F. Silva eds., Parallelism and Implementation of Logic and Constraint Programming. Nova Science Publishers, 1999.

[17] Zs. Németh: Abstract machine design on a multithreaded architecture. Future Generation Computer Systems, Vol. 16, No. 6, Elsevier, 2000, pp. 705-716.

[18] Zs. Németh: A Novel Execution Model for LOGFLOW on a Hybrid Multithreaded Architecture. Proceedings of DAPSYS 2000, Kluwer, 2000, pp 117-126.

[19] Zs. Németh: Issues of a Distributed Parallel Prolog System on Hybrid Multithreaded Architectures. PhD Thesis, Budapest University of Technology and Economics, 2001.

[20] B. Robič, J. Šilc, T. Ungerer: Beyond Dataflow. Journal of Computing and Information Technology, Vol. 8, No. 2, University Computing Centre, Zagreb, 2000, pp. 89-101.

[21] G. Schellhorn: Verification of Abstract State Machines. PhD Thesis, University of Ulm, 1999.

[22] D.Sima, T. Fountain, P. Kacsuk: Advanced Computer Architectures. Addison Wesley, 1997.

[23] R. Stärk, J. Schmid, E. Börger: Java and the Java Virtual Machine. Springer, 2001

[24] H.Tomiyasu, T. Kawano, R. Taniguchi, M. Amamiya: KUMP/D: the Kyushu University Multimedia Processor. Proceedings of the Computer Architectures for Machine Perception, CAMP'95, IEEE Computer Society Press, 1995, pp. 367-374.

Appendix A: ASM code for the Logicflow model (ASM1)

1 A DO token on the request.in arc of a Unit node

```
if node(loc(t)) = UNIT & type(t) = DO
then
   extend STREAM by s with
     stream(parent(loc(t)), returnport(loc(t)), colour(t)) := s
     seq i = 1..n
         let θ = mgu(act(t), head(nth(predicate(loc(t)), i)))
         if θ! = nil then
            extend TOKEN by t' with
                   colour(t') := colour(t)
                   loc(t') := parent(loc(t))
                   subst(t') := subst(t)@θ
                   type(t') := SUCC
                   instream(s, t') := true
            endextend
         endif
     endseq
```

```
    loc(t) := parent(loc(t))
    type(t) := FAIL
    subst(t) := {}
    instream(s,t) := true
  endextend
where n = length(predicate(loc(t)))
```

2 A SUB token on the request.in arc of an And node

2a First appearance

```
if node(loc(t)) = AND & type(t) = SUB & mode(loc(t)) = create
then
  extend NODE by n with
    parent(n) := loc(t)
    child(loc(t)) := n
    mode(n) := create
    returnport(n) := reply.in
    predicate(n) := procdef(goal(loc(t)))
    loc(t) := n
  endextend
  if counter(loc(t), colour(t)) = undef
    then
      counter(loc(t), colour(t)) := 1
    else
      counter(loc(t), colour(t)) := counter(loc(t), colour(t)) + 1
  endif
  if and_state(loc(t), colour(t)) = undef
    then
      and_state(loc(t), colour(t)) := open
  endif
  instream(stream(loc(t), request.in, colour(t)), t) := false
  mode(loc(t)) := active
  act(t) := goal(loc(t))
  type(t) := DO
```

6.1 2b Further appearances

```
if node(loc(t)) = AND & type(t) = SUB & mode(loc(t)) = active
then
  if counter(loc(t), colour(t)) = undef
    then
      counter(loc(t), colour(t)) := 1
    else
      counter(loc(t), colour(t)) := counter(loc(t), colour(t)) + 1
  endif
  instream(stream(loc(t), request.in, colour(t)), t) := false
  loc(t) := child(loc(t))
  act(t) := goal(loc(t))
  type(t) := DO
```

3 A FAIL2 token on the request.in arc of an And node

```
if node(loc(t)) = AND
   & type(t) = FAIL2
   & mode(loc(t)) = active
   & card(stream(loc(t), request.in, colour(t))) = 1
   then
      instream(stream(loc(t), request.in, colour(t)), t) := false
      and_state(loc(t), colour(t)) := closed
      if counter(loc(t), colour(t)) = 0 | counter(loc(t), colour(t)) = undef
         then
            instream(stream(next(loc(t)), request.in, colour(t)), t) := true
            loc(t) := next(loc(t))
         else
            TOKEN(t) := false
      endif
```

4 A SUCC token on the reply.in arc of an And node

```
if node(loc(t)) = AND & type(t) = SUCC & mode(loc(t)) = active
then
   if node(next(loc(t))) = UNIFY
      then let port = reply.in
      else let port = request.in
   endif
   if stream(next(loc(t)), reply.in, colour(t)) = undef
   then
      extend STREAM by s with
         stream(next(loc(t)), port, colour(t)) := s
         instream(s, t) := true
      endextend
   else
      instream(stream(next(loc(t)), port, colour(t)), t) := true
   endif
   instream(stream(loc(t), request.in, colour(t)), t) := false
   loc(t) := next(loc(t))
   type(t) := SUB
```

5 A FAIL token on the reply.in arc of an And node when it is open

```
if node(loc(t)) = AND
   & type(t) = FAIL
   & mode(loc(t)) = active
   & card(stream(loc(t), reply.in, colour(t))) = 1
   & and_state(loc(t), colour(t)) = open
   then
      counter(loc(t), colour(t)) := counter(loc(t), colour(t)) - 1
      TOKEN(t) := false
```

6 A FAIL token on the reply.in arc of an And node when it is closed

```
if node(loc(t)) = AND
```

```
& type(t) = FAIL
& mode(loc(t)) = active
& card(stream(loc(t), reply.in, colour(t)))) = 1
& and_state(loc(t), colour(t)) = closed
then
   if node(next(loc(t))) = UNIFY
   then let port = reply.in
   else let port = request.in
   endif
   counter(loc(t), colour(t)) := counter(loc(t), colour(t)) − 1
   instream(stream(loc(t), reply.in, colour(t)), t) := false
   if counter(loc(t), colour(t)) = 0
   then
      instream(stream(next(loc(t)), port, colour(t)), t) := true
      loc(t) := next(loc(t))
   endif
```

7 A DO token on the request.in arc of a Unify node

7a First appearance

```
if node(loc(t)) = UNIFY & type(t) = DO & mode(loc(t)) = create
then
   let θ = mgu(act(t), head(predicate(loc(t))))
   if θ! = nil
   then
      extend COLOUR by newcolour with
         colour(t) := newcolour
         colour_context(loc(t), newcolour) := colour(t)
         subst_context(loc(t), newcolour) := subst(t)@θ
         extend NODE by n1, n2, ...nm with
            node(ni) := AND
            mode(ni) := create
            first(loc(t)) := n1
            prev(n1) := loc(t)
            last(loc(t)) := nm
            next(nm) := loc(t)
            prev(nk) := nk−1
            next(nk) := nk+1
            goal(ni) := nth(body(predicate(loc(t))), i)
            loc(t) := n1
            type(t) := SUB
            subst(t) := θ
            extend STREAM by s with
               stream(first(loc(t)), request.in, newcolour) := s
               extend TOKEN by t' with
                  loc(t') := n1
                  type(t') := FAIL2
                  colour(t') := newcolour
                  instream(s, t') := true
               endextend
               instream(s, t) := true
            endextend
         endextend
         mode(loc(t)) := active
      endextend
```

```
   else
      extend STREAM by s with
         stream(parent(loc(t)), returnport(loc(t)), colour(t)) := s
         type(t) := FAIL
         instream(s, t) := true
         loc(t) := parent(loc(t))
      endextend
   endif
   where m = length(body(predicate(loc(t)))), 1 ≤ i ≤ m, 1 < k < m
```

7b Further appearances

```
if node(loc(t)) = UNIFY & type(t) = DO & mode(loc(t)) = active
then
   let θ = mgu(act(t), head(predicate(loc(t))))
   if θ! = nil
   then
      extend COLOUR by newcolour with
         colour(t) := newcolour
         colour_context(loc(t), newcolour) := colour(t)
         subst_context(loc(t), newcolour) := subst(t)@θ
         extend STREAM by s with
            stream(first(loc(t)), request.in, newcolour) := s
            loc(t) := first(loc(t))
            type(t) := SUB
            instream(s, t) := true
            subst(t) := θ
            extend TOKEN by t' with
               loc(t') := first(loc(t))
               type(t') := FAIL2
               colour(t') := newcolour
               instream(s, t') := true
            endextend
         endextend
      endextend
   else
      extend STREAM by s with
         stream(parent(loc(t)), returnport(loc(t)), colour(t)) := s
         type(t) := FAIL
         instream(s, t) := true
         loc(t) := parent(loc(t))
      endextend
   endif
```

8 A SUB token on the reply.in arc of a Unify node

```
if node(loc(t)) = UNIFY & type(t) = SUB
then
   instream(stream(loc(t), reply.in, colour(t)), t) := false
   colour(t) := saved_colour
   subst(t) := saved_subst@subst(t)
   type(t) := SUCC
   loc(t) := parent(loc(t))
   if stream(parent(loc(t)), returnport(loc(t)), saved_colour) = undef
   then
```

```
        extend STREAM by s with
            stream(parent(loc(t)), returnport(loc(t)), saved_colour) := s
            instream(s, t) := true
        endextend
    else
        instream(stream(parent(loc(t)), returnport(loc(t)), saved_colour), t) := true
    endif
where saved_colour ≡ colour_context(loc(t), colour(t)), saved_subst ≡ subst_context(loc(t), colour(t))
```

9 A FAIL token on the reply.in arc of a Unify node

```
if node(loc(t)) = UNIFY & type(t) = FAIL2 & card(stream(loc(t), reply.in, colour(t))) = 1
then
    let saved_colour = colour_context(loc(t), colour(t))
    instream(stream(loc(t), reply.in, colour(t)), t) := false
    colour(t) := saved_colour
    type(t) := FAIL
    loc(t) := parent(loc(t))
    if stream(parent(loc(t)), returnport(loc(t)), saved_colour) = undef
    then
        extend STREAM by s with
            stream(parent(loc(t)), returnport(loc(t)), saved_colour) := s
            instream(s, t) := true
        endextend
    else
        instream(stream(parent(loc(t)), returnport(loc(t)), saved_colour), t) := true
    endif
```

10 A DO token on the request.in arc of an Or node

10a First appearance

```
if node(loc(t)) = OR & type(t) = DO & mode(loc(t)) = create
then
    extend COLOUR by newcolour with
        colour(t) := newcolour
        colour_context(loc(t), newcolour) := colour(t)
        extend NODE by n1, n2 with
            child1(loc(t)) := n1
            child2(loc(t)) := n2
            parent(n1) := loc(t)
            parent(n2) := loc(t)
            returnport(n1) := reply.in1
            returnport(n2) := reply.in2
            if k = 1
            then
                predicate(n1) := car(predicate(loc(t)))
                predicate(n2) := cdr(predicate(loc(t)))
            else
                predicate(n1) := [clause1, ...clausek-1]
                predicate(n2) := [clausek, ...clausen]
            endif
            extend TOKEN by t' with
                loc(t') := n2
                type(t') := DO
```

```
            colour(t') := newcolour
            subst(t') := subst(t)
         endextend
         loc(t) := n1
      endextend
   endextend
   mode(loc(t)) := active
   where clause_i ≡ nth(predicate(loc(t)), i), k = min{i|body(clause_i) ≠ nil}
```

10b Further appearances

```
if node(loc(t)) = OR & type(t) = DO & mode(loc(t)) = active
then
   extend COLOUR by newcolour with
      colour(t) := newcolour
      colour_context(loc(t), newcolour) := colour(t)
      extend TOKEN by t' with
         loc(t') := child2(loc(t))
         type(t') := DO
         colour(t') := newcolour
         subst(t') := subst(t)
      endextend
      loc(t) := child1(loc(t))
   endextend
endextend
```

11 A SUCC token on any of the reply.in arcs of an Or node

```
if node(loc(t)) = OR & type(t) = SUCC
then
   let saved_colour = colour_context(loc(t), colour(t))
   colour(t) := saved_colour
   loc(t) := parent(loc(t))
   if instream(stream(loct(t), reply.in1, colour(t), t) = true
   then
      instream(stream(loct(t), reply.in1, colour(t), t) := false
   else
      instream(stream(loct(t), reply.in2, colour(t), t) := false
   endif
   if stream(parent(loc(t)), returnport(loc(t)), saved_colour) = undef
   then
      extend STREAM by s with
         stream(parent(loc(t)), returnport(loc(t)), saved_colour) := s
         instream(s, t) := true
      endextend
   else
      instream(stream(parent(loct(t)), returnport(loc(t)), saved_colour), t) := true
   endif
```

12 A FAIL token on any of the reply.in arcs of an Or node

12a The Or node is in wait1 state

```
if node(loc(t)) = OR
```

```
    & type(t) = FAIL
    & or_state(loc(t), colour(t)) = wait1
    & (∃s ∈ STREAM : instream(s,t) & card(s) = 1)
    then
        or_state(loc(t), colour(t)) := wait2
        TOKEN(t) := false
        STREAM(s) := false
```

12b The Or node is in wait2 state

```
if node(loc(t)) = OR
& type(t) = FAIL
& or_state(loc(t), colour(t)) = wait2
& (∃z ∈ STREAM : instream(z,t) & card(z) = 1)
then
    let saved_colour = colour_context(loc(t), colour(t))
    if stream(parent(loc(t)), returnport(loc(t)), saved_colour) = undef
    then
        extend STREAM by s with
            stream(parent(loc(t)), returnport(loc(t)), saved_colour) := s
            instream(s,t) := true
        endextend
    else
        instream(stream(parent(loct(t)), returnport(loc(t)), saved_colour), t) := true
    endif
    STREAM(z) := false
    colour(t) := saved_colour
    loc(t) := parent(loc(t))
```

13 A DO token on the request.in arc of an undefined node (child nodes of And and Or nodes are undefined)

```
if node(loc(t)) = undef & type(t) = DO
then
    if length(predicate(loc(t)) = 1
    thenif body(clause_1) ≠ nil
        then
            node(loc(t)) := UNIFY
        else
            node(loc(t)) := UNIT
        endif
    elseif ∀i : body(clause_i) = nil
        then
            node(loc(t)) := UNIT
        else
            node(loc(t)) := OR
        endif
    endif
    where clause_i ≡ nth(predicate(loc(t)), i)
```

14 A DO token on a Query node

```
if node(loc(t)) = QUERY & type(t) = DO
then
```

```
extend NODE by n with
    child(loc(t)) := n
    parent(n) := loc(t)
    predicate(n) := procdef(act(t))
    mode(n) := create
    returnport(n) := reply.in
    loc(t) := n
endextend
```

15 A SUCC token on a Query node

```
if node(loc(t)) = QUERY & type(t) = SUCC
then
    TOKEN(t) := false
```

16 A FAIL token on a Query node

```
if node(loc(t)) = QUERY & type(t) = FAIL & card(stream(loc(t), reply.in, colour(t))) = 1
then
    TOKEN(t) := false
    STREAM(stream(loc(t), reply.in, colour(t))) := false
```

Appendix B: ASM code for the modified Logicflow model (ASM2)

```
ancestor : NODE → NODE
counter : NODE × COLOUR → INT
```

17 A DO token on the request.in arc of a Unit node

Same as Rule 1

18 A SUB token on the request.in arc of an And or Last_And node

6.2 18a First appearance

```
if node(loc(t)) = (AND|LAST_AND) & type(t) = SUB & mode(loc(t)) = create
then
    extend NODE by n with
        parent(n) := loc(t)
        child(loc(t)) := n
        mode(n) := create
        returnport(n) := reply.in
        predicate(n) := procdef(goal(loc(t)))
        loc(t) := n
    endextend
    instream(stream(loc(t), request.in, colour(t)), t) := false
    mode(loc(t)) := active
    act(t) := goal(loc(t))
    type(t) := DO
```

18b Further appearances

```
if node(loc(t)) = (AND|LAST_AND) & type(t) = SUB & mode(loc(t)) = active
then
   instream(stream(loc(t), request.in, colour(t)), t) := false
   loc(t) := child(loc(t))
   act(t) := goal(loc(t))
   type(t) := DO
```

19 A SUCC token on the reply.in arc of an And node

```
if node(loc(t)) = AND & type(t) = SUCC & mode(loc(t)) = active
then
   if stream(next(loc(t)), request.in, colour(t)) = undef
   then
      extend STREAM by s with
         stream(next(loc(t)), request.in, colour(t)) := s
         instream(s, t) := true
      endextend
   else
      instream(stream(next(loc(t)), request.in, colour(t)), t) := true
   endif
   instream(stream(loc(t), reply.in, colour(t)), t) := false
   counter(ancestor(loc(t), colour(t)) := counter(ancestor(loc(t), colour(t)) + 1
   loc(t) := next(loc(t))
   type(t) := SUB
```

20 A SUCC token on the reply.in arc of a Last_And node

```
if node(loc(t)) = LAST_AND & type(t) = SUCC & mode(loc(t)) = active
then
   if stream(next(loc(t)), reply.in, colour(t)) = undef
   then
      extend STREAM by s with
         stream(next(loc(t)), reply.in, colour(t)) := s
         instream(s, t) := true
      endextend
   else
      instream(stream(next(loc(t)), reply.in, colour(t)), t) := true
   endif
   instream(stream(loc(t), reply.in, colour(t)), t) := false
   loc(t) := next(loc(t))
   type(t) := SUB
```

21 A FAIL token on the reply.in arc of an And or Last_And node

```
if node(loc(t)) = (AND|LAST_AND)
   & type(t) = FAIL
   & mode(loc(t)) = active
   & card(stream(loc(t), reply.in, colour(t))) = 1
   then
      counter(ancestor(loc(t)), colour(t)) := counter(ancestor(loc(t)), colour(t)) - 1
      TOKEN(t) := false
```

22 A DO token on the request.in arc of a Unify node

22a First appearance

```
if node(loc(t)) = UNIFY & type(t) = DO & mode(loc(t)) = create
then
   let θ = mgu(act(t), head(predicate(loc(t))))
   if θ! = nil
   then
      extend COLOUR by newcolour with
         colour(t) := newcolour
         colour_context(loc(t), newcolour) := colour(t)
         subst_context(loc(t), newcolour) := subst(t)@θ
         counter(loc(t), newcolour) := 1
         extend NODE by n1, n2, ...nm with
            if i < m then
                        node(ni) := AND
            else
                        node(ni) := LAST_AND
            endif
            mode(ni) := create
            ancestor(ni) := loc(t)
            first(loc(t)) := n1
            prev(n1) := loc(t)
            last(loc(t)) := nm
            next(nm) := loc(t)
            prev(nk) := nk-1
            next(nk) := nk+1
            goal(ni) := nth(body(predicate(loc(t))), i)
            loc(t) := n1
            type(t) := SUB
            subst(t) := θ
            extend STREAM by s with
               stream(first(loc(t)), request.in, newcolour) := s
               instream(s, t) := true
            endextend
         endextend
         mode(loc(t)) := active
      endextend
   else
      extend STREAM by s with
         stream(parent(loc(t)), returnport(loc(t)), colour) := s
         type(t) := FAIL
         instream(s, t) := true
         loc(t) := parent(loc(t))
      endextend
   endif
where m = length(body(predicate(loc(t)))), 1 ≤ i ≤ m, 1 < k < m
```

22b Further appearances

```
if node(loc(t)) = UNIFY & type(t) = DO & mode(loc(t)) = active
then
   let θ = mgu(act(t), head(predicate(loc(t))))
   if θ! = nil
   then
```

```
      extend COLOUR by newcolour with
         colour(t) := newcolour
         colour(loc(t), newcolour) := colour(t)
         subst(loc(t), newcolour) := subst(t)@θ
         counter(loc(t), newcolour) := 1
         extend STREAM by s with
            stream(first(loc(t)), request.in, newcolour) := s
            loc(t) := first(loc(t))
            type(t) := SUB
            instream(s, t) := true
            subst(t) := θ
         endextend
      endextend
   else
      extend STREAM by s with
         stream(parent(loc(t)), returnport(loc(t)), colour) := s
         type(t) := FAIL
         instream(s, t) := true
         loc(t) := parent(loc(t))
      endextend
   endif
```

23 A SUB token on the reply.in arc of a Unify node

Same as Rule 8

24 The counter of a Unify node is 0

```
if (∃Node, Colour : Node ∈ NODE, Colour ∈ COLOUR) : node(Node) = UNIFY
   & counter(Node, Colour) = 0
   & card(stream(Node, reply.in, Colour)) = 0
   then
      let saved_colour = colour_context(Node, Colour)
      extend TOKEN by t' with
         colour(t') := saved_colour
         type(t') := FAIL
         loc(t') := parent(Node)
         if stream(parent(Node), returnport(Node), saved_colour) = undef
         then
            extend STREAM by s with
               stream(parent(Node), returnport(Node), saved_colour) := s
               instream(s, t') := true
            endextend
         else
            instream(stream(parent(Node), returnport(Node), saved_colour), t') := true
         endif
      endextend
```

25 A DO token on the request.in arc of an Or node

25a First appearance

Same as Rule 10a

25b Further appearances

Same as Rule 10b

26 A SUCC token on any of the reply.in arcs of an Or node

Same as Rule 11

27 A FAIL token on any of the reply.in arcs of an Or node

27a The Or node is in wait1 state

Same as Rule 12a

27b The Or node is in wait2 state

Same as Rule 12b

28 A DO token on the request.in arc of an undefined node (child nodes of And and Or nodes are undefined)

Same as Rule 13

29 A DO token on a Query node

Same as Rule 14

30 A SUCC token on a Query node

Same as Rule 15

31 A FAIL token on a Query node

Same as Rule 16

Acta Cybernetica **15** (2001) 457–465.

Properties of Composite of Closure Operations and Choice Functions

Nghia D. Vu* and Bina Ramamurthy*

Abstract

The equivalence of the family of FDs is among many hottest topics that get a lot of attention and consideration currently. There are many equivalent descriptions of the family of FDs. The closure operation and choice function are two of them. Major results of this paper are the properties of the composite function of the choice functions and closure operations. The first parts of this paper address the theories of the composite function of two choice functions and the sufficient and necessary condition of a composite function of two choice functions to be a choice function. Rest of the paper addresses the sufficient and necessary condition of a composite function of more than two choice functions to be a choice function and a composite function of more than two closure operations to be a closure operation.

Keywords: composite function, choice function, closure operation.

1 Introduction

Equivalent descriptions of the family of functional dependencies (FDs) have been widely studied. Based on the equivalent descriptions, we can obtain many important properties of the family of FDs. Choice function and closure operation are two of many equivalent descriptions of the family of FDs. In this paper, we mostly investigate the choice functions. We show some properties of choice functions, and focus on the comparison between and composite function of two, and more than two choice functions. At the end of this paper, we show a theory of the composite function of two and more than two closure operations.

The results of this paper are divided into four parts. First, some properties of the composite function of two choice functions appear in Section 2. Section 3 presents the results about the composite function of more than two choice functions, and that of more than two closure operations. In the conclusion section, we introduce our plans for future research.

*Department of Computer Science and Engineering, 201 Bell Hall, University at Buffalo, Buffalo, NY 14260. Phone: (716)-645-3180(108). Fax: (716)-645-3464.
E-mail: nghiavu, bina@cse.buffalo.edu

Let us give some necessary definitions that are used in the next section. Those well-known concepts in relational database given in this section can be found in [1, 2, 3, 4, 5, 6, 8].

Definition 1 *Let $U = \{a_1, ..., a_n\}$ be a nonempty finite set of attributes. A functional dependency is a statement of the form $A \to B$, where $A, B \subseteq U$. The FD $A \to B$ holds in a relation $R = \{h_1, ..., h_m\}$ over U if $\forall\ h_i, h_j \in R$ we have $h_i(a) = h_j(a)$ for all $a \in A$ implies $h_i(b) = h_j(b)$ for all $b \in B$. We also say that R satisfies the FD $A \to B$.*

A family of FDs satisfying Armstrong's Axioms is called an f-family over U. Given a family F of FDs over U, there exits a unique minimal f-family F^+ that contains F. It can be seen that F^+ contains all FDs which can be derived from F by Armstrong Axioms.

A relation scheme s is a pair $< U, F >$, where U is a set of attributes, and F is a set of FDs over U.

Let U be a nonempty finite set of attributes and $P(U)$ its power set. A map $L : P(U) \to P(U)$ is called a closure over U if it satisfies the following conditions:

(1) $A \subseteq L(A)$,
(2) $A \subseteq B$ implies $L(A) \subseteq L(B)$
(3) $L(L(A)) = L(A)$.

Set $L(A) = \{a : A \to \{a\} \in F^+\}$, we can see that L is a closure over U. There is a 1-1 correspondence between closures and f-families on U.

A map $C : P(U) \to P(U)$ is called a choice function, if every $A \in P(U)$, then $C(A) \subseteq A$.
If we assume that $C(A) = U - L(U - A)$ (*), we can easily see that C is a choice function.

The relationship like (*) is considered as a 1-1 correspondence between closures and choice functions, which satisfies the following two conditions:

For every $A, B \subseteq U$,

(1) If $C(A) \subseteq B \subseteq A$, then $C(A) = C(B)$
(2) If $A \subseteq B$, then $C(A) \subseteq C(B)$

We call all of choice functions satisfying those two above conditions special choice functions.

There is a 1-1 correspondence between special choice functions and f-families on U.

We define Γ as a set of all of special choice (SC) functions on U. Now we investigate some properties of those functions.

2 Properties of the SC functions

In this section, we give some results related to the composite function of two choice functions.

Let $f, g \in \Gamma$, and we determine a map k as a composite function of f and g as the following:

$$k(X) = f(g(X)) = f.g(X) = fg(X) \text{ for every } X \subseteq U.$$

Let U be a nonempty finite set of attributes, and $f, g \in \Gamma$. We say that f is smaller than g, denoted as $f \leq g$ or $g \geq f$, if for every $X \subseteq U$ we always have $f(X) \subseteq g(X)$.
The "smaller" relation, $\leq$, satisfies these following properties. For every $f, g, h \in \Gamma$:

1) $f = f$ (Reflexive)
2) If $f \leq g$, and $g \leq f$, then $g = f$. (Symmetric)
3) If $f \leq g$, and $g \leq h$, then $f \leq h$. (Transitive)

Proposition 1 *If $f, g \in \Gamma$, then*

$$\begin{array}{llll} 1) & fg \leq f, & 2) & fg \leq g, \\ 3) & gf \leq f, & 4) & gf \leq g. \end{array}$$

Proof. Since $f, g \in \Gamma$, f and g must be SC functions on U. Therefore, we have $g(X) \subseteq X$ for every $X \subseteq U$, then $f(g(X)) \subseteq f(X)$. And f is a SC function on U; so $f(g(X)) \subseteq g(X)$. So we can conclude that $fg \leq f$ and $fg \leq g$. Similarly, we can easily prove $gf \leq f$ and $gf \leq g$. □

Proposition 2 *If f, h and $g \in \Gamma$ and $f \leq g$, then*

$$\begin{array}{lll} 1) & fh & \leq & gh, \\ 2) & hf & \leq & hg. \end{array}$$

Proof. Because f, g and h are three SC functions and $f \leq g$, we always have $f(h(X)) \subseteq g(h(X))$, for every $X \subseteq U$. Since $f \leq g$, we have $f(X) \subseteq g(X)$. h is a SC function, so we have $h(f(X)) \subseteq h(g(X))$. We can conclude that $fh \leq gh$ and $hf \leq hg$. □

Proposition 3 *If f, g, h and $k \in \Gamma$, and $f \leq g$, and $k \leq h$, then $fk \leq gh$.*

Proof. Assume $f, g, h, k \in \Gamma$ and $f \leq g$, and $k \leq h$. According to Proposition 2, we have $fk \leq gk$ and $gk \leq gh$. Therefore, according to the transitive property, we have $fk \leq gh$. □

Theorem 1 *If $f, g \in \Gamma$, then these following two conditions are equivalence:*

$$\begin{array}{lll} 1) & f & \leq & g, \\ 2) & fg & = & f. \end{array}$$

Proof. $(1 \rightarrow 2)$ Assume $f, g \in \Gamma$ and $f \leq g$. Since f is a SC function, f must satisfies this property: if $f(X) \subseteq Y \subseteq X$, then $f(X) = f(Y)$. Therefore, we have $f \leq g$ or $f(X) \subseteq g(X) \subseteq X$ for every $X \subseteq U$, so $f(g(X)) = f(X)$ or we conclude that $fg = f$.
$(2 \rightarrow 1)$ Assume $f, g \in \Gamma$ and $fg = f$. Since f and g are SC functions, according to Proposition 1, we have $fg \leq g$, but $fg = f$, so we have $f \leq g$. The proof is completed. □

From the Theorem 1, we can easily see that if $f \leq g$, then fg is a SC function (since $fg = f$, and f is a SC function).

Lemma 1 *If $f \in \Gamma$, then $ff = f$.*

Proof. It can be seen easily that Lemma 1 holds directly from the Theorem 1. □

Theorem 2 *Let $f, g \in \Gamma$. A composite function of f and g, denoted as fg, is a SC function if and only if $fgf = fg$:*

$$(fg \text{ is a SC function} \Leftrightarrow fgf = fg).$$

Proof. First, we need to prove that fg is a choice function.
For every $X \subseteq U$, we have $g(X) \subseteq X$ because g is a SC function. And f also is a SC function, so if $g(X) \subseteq X$, then $f(g(X)) \subseteq f(X) \subseteq X$. Therefore, we can conclude that $fg(X) \subseteq X$, in other word, we can say that fg is a choice function. Similarly, we can prove that gf is also a choice function.

Now, we prove that fg is a SC function $\Leftrightarrow fgf = fg$. First, we need to prove the statement: if fg is a SC function, then $fgf = fg$. According to Proposition 1, we have $fg \leq f$. And fg is a SC function, so $fgf = fg$ due to Theorem 1.

Then, we just need to prove that if $fgf = fg$, then fg is a SC function. In other words, we need to prove that if $fgf = fg$, then fg satisfies these two conditions (1) and (2):
If $X \subseteq Y$, then $fg(X) \subseteq fg(Y)$, and if $fg(X) \subseteq Y \subseteq X$, then $fg(X) = fg(Y)$.
When $X \subseteq Y$, we have $g(X) \subseteq g(Y)$ since g is a SC function. And when $g(X) \subseteq g(Y)$, we have $f(g(X)) \subseteq f(g(Y))$ or $fg(X) \subseteq fg(Y)$ since f is also a SC function.

We have $fg(X) \subseteq Y \subseteq X$, so $g(fg(X)) \subseteq g(Y) \subseteq g(X)$ or $gfg(X) \subseteq g(Y) \subseteq g(X)$ since g is a SC function. And since f is also a SC function, we also have $f(gfg(X)) \subseteq f(g(Y)) \subseteq f(g(X))$ or $fgfg(X) \subseteq fg(Y) \subseteq fg(X)$. However, fgf =fg, so that leads to that $fgg(X) = fgfg(X) \subseteq fg(Y) \subseteq fg(X)$. We can rewrite that expression as $fgg(X) \subseteq fg(Y) \subseteq fg(X)$. According to Lemma 1, we have $gg(X) = g(X)$, so $fgg(X) = fg(X) \subseteq fg(Y) \subseteq fg(X)$. Therefore, $fg(X) = fg(Y)$.
Consequently, we can conclude that fg is a SC function iff $fgf = fg$. The proof is completed. □

Theorem 3 *Let $f, g \in \Gamma$. Then fg and gf are simultaneously SC functions if and only if $fg = gf$.*

Proof. In the proof of Theorem 2, already we have proved that fg and gf are always choice functions when f and g are SC functions.
We need to prove this statement: if fg and gf are simultaneously SC functions, then $fg = gf$, for $f, g \in \Gamma$.
According to Proposition 1, we have $fg \leq g$ and $fg \leq f$. So due to Proposition 3, we have $(fg)(fg) \leq gf$. But we also have fg is a SC function, so $(fg)(fg) = fg$ due to Lemma 1. Thus, $(fg)(fg) = fg \leq gf$. Similarly, we also have $gf \leq fg$. Hence, we have $fg \leq gf \leq fg$, so we can conclude that $fg = gf$.
We just need to prove that: if $fg = gf$, then fg and gf are simultaneously SC functions for f,g $\in \Gamma$. In other words, we need to prove that if $fg = gf$, then fg and gf satisfies these two conditions (1) and (2):
If $X \subseteq Y$, then $fg(X) \subseteq fg(Y)$ and $gf(X) \subseteq gf(Y)$.
If $fg(X) \subseteq Y \subseteq X$, then $fg(X) = fg(Y)$, and if $gf(X) \subseteq Y \subseteq X$, then $gf(X) = gf(Y)$.

In the proof of Theorem 2, we have already proved: if $X \subseteq Y$, then $fg(X) \subseteq fg(Y)$. Similarly, we also can prove that $gf(X) \subseteq gf(Y)$.
We have $fg(X) \subseteq Y \subseteq X$, so $g(fg(X)) \subseteq g(Y) \subseteq g(X)$ or $gfg(X) \subseteq g(Y) \subseteq g(X)$ since g is a SC function. And since f is also a SC function, we also have $f(gfg(X)) \subseteq f(g(Y)) \subseteq f(g(X))$ or $fgfg(X) \subseteq fg(Y) \subseteq fg(X)$. However, $fg = gf$, so that leads to that $ffgg(X) = fgfg(X) \subseteq fg(Y) \subseteq fg(X)$. We can rewrite that expression as $ffgg(X) \subseteq fg(Y) \subseteq fg(X)$. According to Lemma 1, we have $gg = g$ and $ff = f$, so $ffgg(X) = fg(X) \subseteq fg(Y) \subseteq fg(X)$. Therefore, $fg(X) = fg(Y)$.
Similarly, we also prove that if $gf(X) \subseteq Y \subseteq X$, then $gf(X) = gf(Y)$.
Consequently, we can say that fg and gf are simultaneously SC functions if and only if $fg = gf$ for $f, g \in \Gamma$. The proof is completed. □

So far, we have covered some properties of the composition of two SC functions and found out some interesting results. However, we would like to raise the following two questions:
Can we generalize the Theorem 2 for the composition of more than two SC functions? Will we get the same answer? More generally, what is a necessary and sufficient condition such that a composite function of more than two SC functions is a SC function?

3 Composite of more than two SC functions and more than two closure operations

In order to generalize the Theorem 2, we first need to observe the composition of three SC functions before we can go any further.

Theorem 4 *Let f, g; and $h \in \Gamma$. A composite function of f, g, and h, denoted as fgh, is a SC function if and only if $fghfg = fgh$:*

$$(fgh \text{ is a SC function} \Leftrightarrow fghfg = fgh)$$

Proof. We can easily prove that fgh is a choice function.
For every $X \subseteq U$, we have $h(X) \subseteq X$ because g is a SC function. And f and g also are SC functions, so if $h(X) \subseteq X$, then $g(h(X)) \subseteq h(X) \subseteq X$, then $f(g(h(X))) \subseteq g(h(X)) \subseteq h(X) \subseteq X$. Therefore, we can conclude that $fgh(X) \subseteq X$, in other word, we can say that fgh is a choice function. Now, we must prove that fgh is a SC function $\Leftrightarrow fghfg = fgh$.

First, we need to prove the statement: if fgh is a SC function, then $fghfg = fgh$.
According to Proposition 1, we have $gh \leq g$ or $g(h(X)) \subseteq g(X)$, for every $X \subseteq U$. And f is a SC function, so $f(g(h(X)) \subseteq f(g(X))$, and $f(g(X)) \subseteq g(X) \subseteq X$. Thus, we have that $f(g(h(X))) \subseteq f(g(X)) \subseteq X$, so we have $f(g(h(f(g(X))))) = f(g(h(X)))$ or $fghfg = fgh$ since fgh is a SC function.

Then, we just need to prove that if $fghfg = fgh$, then fgh is a SC function. In other words, we need to prove that if $fghfg = fgh$, then fgh satisfies these two conditions (1) and (2):
If $X \subseteq Y$, then $fgh(X) \subseteq fgh(Y)$, and if $fgh(X) \subseteq Y \subseteq X$, then $fgh(X) = fgh(Y)$.

When $X \subseteq Y$, we have $h(X) \subseteq h(Y)$ since h is a SC function. And when $h(X) \subseteq h(Y)$, we have $g(h(X)) \subseteq g(h(Y))$ or $gh(X) \subseteq gh(Y)$ since g is a SC function. And since f is also a SC function, we have $f(gh(X)) \subseteq f(gh(Y))$ or $fgh(X) \subseteq fgh(Y)$.

We have $fgh(X) \subseteq Y \subseteq X$, so $h(fgh(X)) \subseteq h(Y) \subseteq h(X)$ or $hfgh(X) \subseteq h(Y) \subseteq h(X)$ since h is a SC function. And since g is also a SC function, we also have $g(hfgh(X)) \subseteq g(h(Y)) \subseteq g(h(X))$ or $ghfgh(X) \subseteq gh(Y) \subseteq gh(X)$. Similarly, we have $fghfgh(X) \subseteq fgh(Y) \subseteq fgh(X)$ since f is a SC function. However, $fghfg = fgh$, so that leads to that $fghfgh(X) = fghh(X) \subseteq fgh(Y) \subseteq fgh(X)$. We can rewrite that expression as $fghh(X) \subseteq fgh(Y) \subseteq fgh(X)$. According to Lemma 1, we have $hh(X) = h(X)$, so $fghh(X) = fgh(X) \subseteq fgh(Y) \subseteq fgh(X)$. Therefore, $fgh(X) = fgh(Y)$.
Consequently, we can conclude that fgh is a SC function iff $fghfg = fgh$. The proof is completed. □

It can be seen easily that we can generalize the Theorem 4 for the composite of more than three SC functions with the result and proof analogous to Theorem 4.

As we used to mention in the Introduction part, there is a relation (*) between the choice function and closure. For every $A \in P(U)$, if we assume that $C(A) = U - L(U - A)(*)$, we can prove that C is a choice function. After investigating some properties of the composite of choice functions, we are willing to show that

the closure operation has similar property. First, we need to give a definition of the composite function of closure operations.

Let $f, g \in L$, a set of all of closure operation on U. We determine a map k as a composite function of f and g as the following:

$$k(X) = f(g(X)) = f.g(X) = fg(X) \text{ for every } X \subseteq U.$$

We have similar definition of the composite function of more than two closure operations.
Here is the result about the composite of closure operations.

Theorem 5 *Let f, g and $h \in L$, a set of all of closure operation on U. A composite function of f, g and h, denoted as fgh, is a closure (or closure operation) if and only if $fghfg = fgh$.*

$$(\textit{That is, } fgh \textit{ is a closure } \Leftrightarrow fghfg = fgh)$$

Proof. First we prove this statement: if f, g, h and fgh are closures, then $fghfg = fgh$.
For every $X \subseteq U$, we have $X \subseteq h(X)$ since h is a closure. From $X \subseteq h(X)$, we have $g(X) \subseteq g(h(X))$ since g is a closure. Similarly, we have $f(g(X)) \subseteq f(g(h(X)))$. Since f is a closure, we have $g(X) \subseteq f(g(X))$. And since g is a closure, we have $X \subseteq g(X)$. Thus, $X \subseteq f(g(X))$. So we can lead to $X \subseteq f(g(X)) \subseteq f(g(h(X)))$. We can rewrite in the other form $X \subseteq fg(X) \subseteq fgh(X)$. Since fgh is a closure, we have $fgh(X) \subseteq fgh(fg(X)) \subseteq fgh(fgh(X))$. Because fgh is a closure, we have $fgh(fgh(X)) = fgh(X)$. Hence $fgh(X) \subseteq fgh(fg(X)) \subseteq fgh(fgh(X)) = fgh(X)$. So we can conclude that $fgh(fg(X)) = fgh(X)$ or $fghfg(X) = fgh(X)$.

Now, we move to prove the reversed statement: if $fghfg = fgh$, then fgh is a closure.
In order to prove fgh is a closure, we need to prove that fgh satisfies those three conditions:

1) $X \subseteq fgh(X)$,
2) $X \subseteq Y$ implies $fgh(X) \subseteq fgh(Y)$, for X and $Y \subseteq U$, and
3) $fgh(fgh(X)) = fgh(X)$.

We have already proved 1) above.
Since h is a closure, from $X \subseteq Y$, we have $h(X) \subseteq h(Y)$. Similarly, we have $g(h(X)) \subseteq g(h(Y))$, then $f(g(h(X))) \subseteq f(g(h(Y)))$ or $fgh(X) \subseteq fgh(Y)$. Thus, fgh satisfies 2).
Since $fghfg = fgh$, we have $fgh(fgh(X)) = fghfgh(X) = fghfg(h(X)) = fgh(h(X)) = fghh(X) = fgh(X)$ since h is a closure, which satisfies the third condition $hh(X) = h(X)$. Therefore, fgh also satisfies three conditions. So fgh is a closure if $fghfg = fgh$. The proof is completed. □

Similarly to the SC function, we can generalize Theorem 5 for the composite of more than three closure operations with analogous result and proof.

4 Open problems

Our further research will be devoted to following open problems:

Open Problem 1. Is the union, intersection, or subtraction of two SC functions a SC function?

Open Problem 2. We would like to apply above results and Theorems into design of algorithm. We have two relation schemes $s =< U, F >$ and $t =< U, V >$, where U is a set of attributes and F and V are two different sets of FDs over U. We define F^+ and V^+ be a set of all FDs that can be derived from F and V respectively. Is it possible build a closure f and a closure g from F^+ and V^+ respectively such that $fg = fgf$? If so, how can we design fg? In other word, how can we design a relation scheme $w =< U, H >$ from which we can build H^+, from which we can design the closure $fg = fgf$? If so, is it possible to generalize this design for more than two closure operations?

Acknowledgments

The authors are grateful to Dr. Thi D. Vu for useful comments to the first version of the manuscripts.

References

[1] Armstrong W.W., Dependency Structures of Database Relationships. Information Processing 74, Holland Publ. Co., 1974, pp. 580-583.

[2] Beeri C., Bernstein P. A., Computational problems related to the design of normal form relation schemes. ACM Trans. on Database Syst. 4, 1, 1979, pp. 30-59.

[3] Beeri C., Dowd M., Fagin R., Staman R., On the Stucture of Armstrong relations for Functional Dependencies. J. ACM 31, 1, 1984, pp. 30-46.

[4] Demetrovics J., Katona G.O.H., A survey of some combinatorial results concerning functional dependencies in database relations. Annals of Mathematics and Artificial Intelligence, 7(1993) 63-82.

[5] Demetrovics J., Thi V.D., Armstrong Relation Functional dependencies and Strong Dependencies. Computers and AI, Vol. 14, 1995, No. 3, pp. 279-298.

[6] Demetrovics J., Thi V.D., Some results about normal forms for functional dependencies in the relational data model. Discrete Applied Mathematics 69, 1996, pp. 61-74.

[7] Demetrovics J., Thi V.D., Describing Candidate Keys by Hypergraphs. Computer and Artificial Intelligence , Vol.18, 1999, No. 2, 191-207.

[8] Ullman D. J., Principle of database and knowledge-based systems. 1989 Computer Science Press.

Received January, 2001

Acta Cybernetica **15** (2002) 467–478.

The Home Marking Problem and Some Related Concepts

Roxana Melinte, Olivia Oanea, Ioana Olga,
and Ferucio Laurenţiu Ţiplea*

Abstract

In this paper we study the *home marking problem* for Petri nets, and some related concepts to it like confluence, noetherianity, and state space inclusion. We show that the home marking problem for inhibitor Petri nets is undecidable. We relate then the existence of home markings to confluence and noetherianity and prove that confluent and noetherian Petri nets have an unique home marking. Finally, we define some versions of the state space inclusion problem related to the home marking and sub-marking problems, and discuss their decidability status.

1 Introduction and Preliminaries

A *home marking* of a system is a marking which is reachable from every reachable marking in the system. The identification of home markings is an important issue in system design and analysis. A typical example is that of an operating system which, at boot time, carries out a set of initializations and then cyclically waits for, and produces, a variety of input/output operations. The states that belong to the ultimate cyclic behavioural component determine the central function of this type of system. The markings modeling such states are the home markings.

The existence of home markings is a widely studied subject in the theory of Petri nets [6, 1, 15, 2, 14, 4, 13], but only for very particular classes of them. Thus, in [1] it has been proven that live and 1-safe free-choice Petri nets have home markings. The result has successively been extended to live and safe free-choice Petri nets [15], live and safe equal-conflict Petri nets [14], and deterministically synchronized sequential process systems [11]. All these results make use, more or less directly, of a *confluence property* which is induced by liveness and safety.

The *home marking problem* for Petri nets (that is, the problem of deciding whether or not a given marking of a Petri net is a home marking) has been proven decidable in [5]. In our paper we show that this problem is undecidable for inhibitor Petri nets (section 2). Then, we relate the concept of a home marking to the

*Faculty of Computer Science "Al. I. Cuza" University 6600 Iaşi, Romania, e-mail: {abur,olivia,olgai,fltiplea}@infoiasi.ro

properties of confluence, safety, and noetherianity, and prove that confluent and noetherian Petri nets have an unique home marking (section 3). In Section 4 we define some versions of the state space inclusion problem for Petri nets, related to the home marking problem, and discuss their decidability status. We close the paper by some conclusions.

The rest of this section is devoted to a short introduction to Petri nets (for details the reader is referred to [12, 9]). A (finite) *Petri net* (with infinite capacities), abbreviated PN, is a 4-tuple $\Sigma = (S, T, F, W)$, where S and T are two finite non-empty sets (of *places* and *transitions*, respectively), $S \cap T = \emptyset$, $F \subseteq (S \times T) \cup (T \times S)$ is the *flow relation*, and $W : (S \times T) \cup (T \times S) \to \mathbf{N}$ is the *weight function* of Σ satisfying $W(x, y) = 0$ iff $(x, y) \notin F$. When all weights are one, Σ is called *ordinary*.

A *marking* of a Petri net Σ is a function $M : S \to \mathbf{N}$. A *marked Petri net* , abbreviated mPN, is a pair $\gamma = (\Sigma, M_0)$, where Σ is a PN and M_0, the *initial marking* of γ, is a marking of Σ.

The behaviour of the net γ is given by the so-called *transition rule*, which consists of:

(a) the *enabling rule*: a transition t is *enabled* at a marking M (in γ), abbreviated $M[t\rangle_\gamma$, iff $W(s, t) \leq M(s)$, for any place s;

(b) the *computing rule*: if $M[t\rangle_\gamma$ then t may *occur* yielding a new marking M', abbreviated $M[t\rangle_\gamma M'$, defined by $M'(s) = M(s) - W(s, t) + W(t, s)$, for any place s.

The transition rule is extended homomorphically to sequences of transitions by $M[\lambda\rangle_\gamma M$, and $M[wt\rangle_\gamma M'$ whenever there is a marking M'' such that $M[w\rangle_\gamma M''$ and $M''[t\rangle_\gamma M'$, where M and M' are markings of γ, $w \in T^*$ and $t \in T$.

Let $\gamma = (\Sigma, M_0)$ be a marked Petri net. A word $w \in T^*$ is called a *transition sequence* of γ if there exists a marking M of γ such that $M_0[w\rangle_\gamma M$. Moreover, the marking M is called *reachable* in γ. The set of all reachable markings of γ is denoted by $[M_0\rangle_\gamma$ (or $[M_0\rangle$ when γ is clear from context).

A Petri net γ is called *n-safe*, where $n \geq 1$ is a natural number, if $M(s) \leq n$ for all reachable marking M; γ is called *safe* if it is n-safe for some n. Clearly, a Petri net is safe iff it has a finite set of reachable markings.

2 The Home Marking Problem

A *home marking* of a system is a marking which is reachable from every reachable marking in the system. For Petri nets, home markings are defined as follows.

Definition 2.1 A marking M of a Petri net $\gamma = (\Sigma, M_0)$ is called a *home marking* of γ if $M \in [M'\rangle$ for all $M' \in [M_0\rangle$.

The Home Marking Problem (HMP)
Instance: $\gamma = (\Sigma, M_0)$ and a marking $\overline{M}$ of γ;
Question: is $\overline{M}$ a home marking of γ ?

In [5], *home spaces* of Petri nets are considered. A home space of a Petri net γ is any set HS of markings of γ such that for any reachable marking M there is a marking $M' \in HS$ reachable from M. If HS is singleton, its unique element is a home marking.

A set A of markings of a Petri net γ is called *linear* if there are a marking M of γ and a finite set $\{M_1, \ldots, M_n\}$ of markings of γ such that

$$(\forall M' \in A)(\forall 1 \leq i \leq n)(\exists k_i \in \mathbf{N})(M' = M + \sum_{i=1}^{n} k_i M_i).$$

The main result proved in [5] states that it is decidable whether or not a linear set of markings is a home space. Therefore, the home marking problem is decidable because any singleton set is linear.

The concept of a home marking can also be considered for extended Petri nets (like inhibitor, reset etc.) by taking into consideration their transition relation. In what follows we show that it is undecidable whether or not a marking of an inhibitor Petri net is a home marking. First, recall the concepts of an inhibitor net and counter machine.

A *k-inhibitor net* ($k \geq 0$) is a couple $\gamma = (\Sigma, I)$, where Σ is a net and I is a subset of $S \times T$ such that $F \cap I = \emptyset$ and $|\{s \in S | (s,t) \in I\}| \leq k$ for all $t \in T$.

Let $\gamma = (\Sigma, I)$ be an inhibitor net, M a marking of γ and $t \in T$. Then,

$$M[t\rangle_{\gamma,i} \Leftrightarrow M[t\rangle_{\Sigma} \wedge (\forall s \in S)((s,t) \in I \Rightarrow M(s) = 0),$$

and

$$M[t\rangle_{\gamma,i} M' \Leftrightarrow M[t\rangle_{\gamma,i} \wedge M[t\rangle_{\Sigma} M'.$$

A *deterministic counter machine* (DCM) is a 6-tuple $A = (Q, q_0, q_f, C, x_0, I)$, where:

(1) Q is a finite non-empty set of *states*, $q_0 \in Q$ is the *initial state*, and $q_f \in Q$ is the *final state*;

(2) C is a finite non-empty set of *counters*. Each counter can store any natural number, and $x_0 : C \to \mathbf{N}$ is the initial content of the counters;

(3) I is a finite set of *instructions*. For each state there is exactly an instruction that can be executed in that state; for q_f there is no instruction. An instruction for a state q is of the one of the following forms:

- *increment instruction* - $I(q, c, q')$

  q : begin
  $c := c + 1$;
  go to q'
  end.

- *test instruction* - $I(q, c, q', q'')$

q : if $c = 0$ then go to q'
else begin
$c := c - 1$;
go to q''
end.

Let $A = (Q, q_0, q_f, C, x_0, I)$ be a DCM. A *configuration* of A is a pair (q, x), where $q \in Q$ and $x : C \to \mathbf{N}$. A configuration (q, x) is called *initial* when $q = q_0$ and $x = x_0$; a configuration (q, x) is called *final* when $q = q_f$.

Let $A = (Q, q_0, q_f, C, x_0, I)$ be a DCM. Define the binary relation $\vdash_A$ on the configurations of A by:

$$(q, x) \vdash_A (q', x') \quad \text{iff one of the following holds:}$$

(1) there is an increment instruction $I(q, c, q')$ such that $x'(c) = x(c) + 1$ and $x'(c') = x(c')$, $\forall c' \in C - \{c\}$;

(2) there is a test instruction $I(q, c, q_1, q_2)$ such that

(2.1) if $x(c) = 0$, then $q' = q_1$ and $x' = x$;

(2.2) if $x(c) \neq 0$, then $q' = q_2$, $x'(c) = x(c) - 1$ and $x'(c') = x(c')$ for all $c' \in C - \{c\}$.

The Halting Problem for counter machines is to decide whether or not a given DCM reaches a final configuration. It is well-known that this problem is undecidable [10].

Theorem 2.1 The home marking problem for 1-inhibitor Petri nets is undecidable.

Proof We show that the halting problem for DCM can be reduced to the home marking problem for 1-inhibitor Petri nets.

Let $A = (Q, q_0, q_f, C, x_0, I)$ be a DCM. Define an 1-inhibitor Petri net as follows:

- to each $u \in Q \cup C$ we associate a place s_u;
- to each increment instruction $I(q, c, q')$ we associate a transition t as in Figure 1(a), and to each test instruction $I(q, c, q', q'')$ we associate two transitions t' and t'' as in Figure 1(b).

A configuration $\sigma = (q, x)$ of A is simulated by the marking M given by:

$$\begin{aligned} M_\sigma(s_q) &= 1, \\ M_\sigma(s_{q'}) &= 0, \quad \forall q' \in Q - \{q\}, \\ M_\sigma(s_c) &= x(c), \quad \forall c \in C. \end{aligned}$$

Let M_0 be the marking corresponding to the initial configuration, and J be the set of pairs (s_c, t'), where s_c and t' are as in Figure 1(b).

The net $\gamma = (\Sigma, J, M_0)$ is an 1-inhibitor net, and we have:

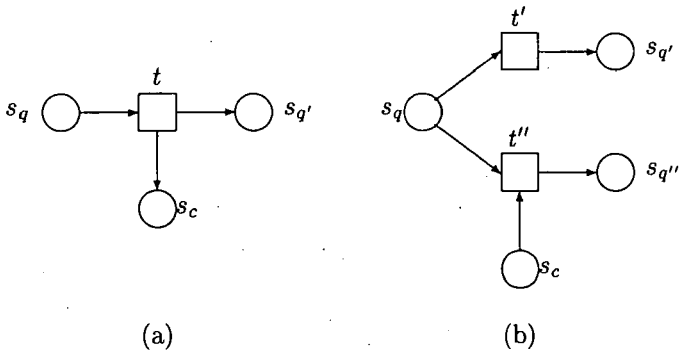


Figure 1: (a) The case $I(q, c, q')$; (b) The case $I(q, c, q', q'')$

(*) $\sigma = (q, x)$ is reachable in A from $\sigma_0 = (q_0, x_0)$ iff M_σ is reachable in γ from M_0.

Modify now the net γ as in Figure 2 (all places and transitions of γ are pictorially represented in the dashed box labelled by γ; the place s^* and the other transitions are new and specific to γ_1).

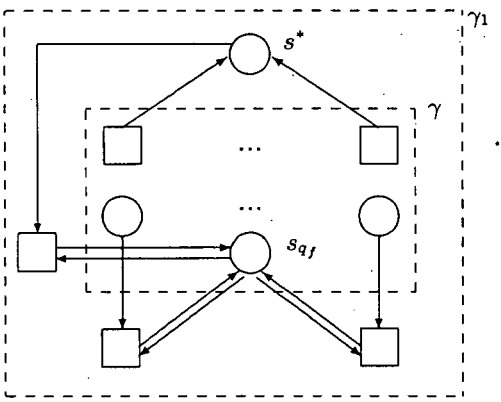


Figure 2: An inhibitor net instance associated to a DCM instance

We prove that A halts iff γ_1 has a home marking. Assume first that A halts, and let (q_f, x) be the final configuration when A halts. Then, $M_{(q_f,x)}(s_{q_f}) = 1$. Therefore, the newly added transitions can be applied yielding the marking $(1, 0, \ldots, 0)$ which is a home marking of γ_1 (this marking can be reached from any reachable marking of γ_1 via the marking $M_{(q_f,x)}$).

Conversely, assume that γ_1 has home markings but A does not halt. Let M be a home marking of γ_1. Then, $M(s_{q_f}) = 0$ (otherwise, A halts). Now we can easily see that the place s^* will be arbitrarily marked (each transition in A induces

a transition in γ_1 which increases by one the place s^*) without the posibility to remove tokens from it because $M(s_{q_f}) = 0$. Therefore, M can not be reached from all reachable markings of γ_1, contradicting the fact that M is a home marking of γ_1. □

3 Confluent and Noetherian Petri Nets

A Petri net is *confluent* if its firing relation is confluent, i.e., for any two reachable markings there is a marking reachable from both of them. This concept proved to be of great importance when we are dealing with the set of reachable markings of a Petri net. It has been considered explicitly for the first time, in connection with Petri nets, in [1], where it has been called *directedness.*

Definition 3.1 An mPN $\gamma = (\Sigma, M_0)$ is *confluent* if $[M_1\rangle \cap [M_2\rangle \neq \emptyset$ for all $M_1, M_2 \in [M_0\rangle$.

Directly from definitions we obtain the following result.

Theorem 3.1 If an mPN has a home marking then it is confluent.

The converse of Theorem 3.1 does not hold generally. For example, the Petri net in Figure 3 is confluent but it does not have any home marking. In case of safe

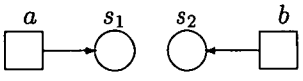


Figure 3: A confluent net which does not have a home marking

Petri nets, the confluence property implies the existence of home markings.

Theorem 3.2 A safe mPN has a home marking iff it is confluent.

The proof of Theorem 3.2 is identical to the proof of Lemma 8.3 in [4] for ordinary Petri nets.

The concept of a *noetherian relation* is another very important concept in the theory of binary relations. As for the confluence property, a Petri net is called *noetherian* if its firing relation is noetherian.

Definition 3.2 An mPN is called *noetherian* if it does not have infinite transition sequences.

Theorem 3.3 Any confluent and noetherian marked Petri net has an unique home marking.

Proof Let $\gamma = (\Sigma, M_0)$ be a confluent and noetherian mPN. Since γ is noetherian, there is a marking $M' \in [M_0\rangle$ such that $\neg(M'[t))$, for any transition t. We will show that M' is the unique home state of γ.

For every reachable marking M of γ the confluence property leads to the existence of a marking M'' such that $M'' \in [M\rangle \cap [M'\rangle$. Then, the property of M' leads to the fact that $M'' = M'$. Therefore, $M' \in [M\rangle$ which shows that M' is the unique home marking of γ. □

Using the coverability tree of a Petri net [12, 9] we can easily prove that the noetherianity property is decidable.

Theorem 3.4 It is decidable whether an mPN is noetherian or not.

Proof An mPN γ is noetherian iff for any leaf node v of the coverability tree of γ, the label of v has no other occurrence on the path from the root to v. Since the coverability tree of a Petri net is always finite and can effectively be constructed, the property of being noetherian is decidable. □

Let us denote by $\mathcal{C}$ ($\mathcal{N}$, $\mathcal{H}$, $\mathcal{H}^*$, $\mathcal{S}$) the class of confluent (noetherian, having home markings, having an unique home marking, safe). It is easily seen that any noetherian mPN has a finite set of reachable markings (equivalently, it is a safe net). The converse of this statement does not hold generally as we can easily see from the net in Figure 4(a). A pictorial view of the relationships between these

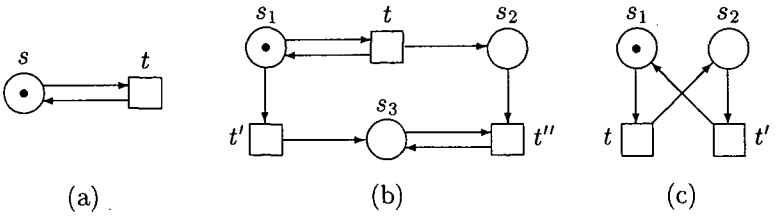


Figure 4: (a) $\gamma \in \mathcal{S} \cap \mathcal{H}^* - \mathcal{N}$; (b) $\gamma \in \mathcal{H}^* - \mathcal{S}$; (c) $\gamma \in \mathcal{S} \cap \mathcal{H} - \mathcal{H}^*$

classes of nets can be found in Figure 5. Some strict inclusions follow from the examples in Figure 4, and some of them are rather trivial.

It is important to know which nets are confluent. In [1] it has been proved that live and 1-safe free-choice Petri nets are confluent. The result has been extended in [15] to live and safe free-choice Petri nets. Further, Recalde and Silva proved in [14] that live and safe equal-conflict Petri nets have home markings (therefore, they are confluent), and the result has been extended to deterministically synchronized sequential process systems in [11].

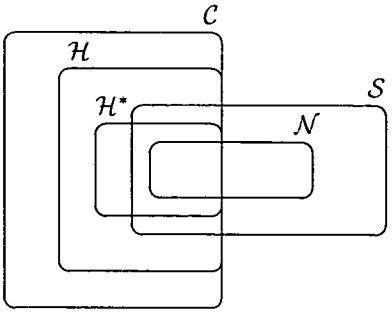


Figure 5: Relationships between classes of Petri nets

4 Home Markings and State Space Inclusions

The home marking problem can be naturally related to some particular versions of the space inclusion problem for Petri nets [7]. In order to define them we need first the following concept.

Definition 4.1 Let $\gamma = (\Sigma, M_0)$ be a mPN and $\overline{M}$ a marking of γ. The *dual of* γ *w.r.t.* $\overline{M}$, denoted by $\overline{\gamma}$, is the Petri net defined as follows:

- $\overline{\gamma} = (\overline{\Sigma}, \overline{M})$;
- $\overline{\Sigma} = (S, \overline{T}, \overline{F}, \overline{W})$;
- $\overline{T} = \{\bar{t} | t \in T\}$;
- $(s, \bar{t}) \in \overline{F}$ iff $(t, s) \in F$, for all $s \in S$ and $t \in T$, and
  $(\bar{t}, s) \in \overline{F}$ iff $(s, t) \in F$, for all $s \in S$ and $t \in T$;
- $\overline{W}(s, \bar{t}) = W(t, s)$ and $\overline{W}(\bar{t}, s) = W(s, t)$, for all $s \in S$ and $t \in T$.

For a sequence $u = t_1 \cdots t_n$ of transitions of a Petri net Σ denote by $\overline{u}$ the sequence $\overline{u} = \bar{t}_n \cdots \bar{t}_1$.

Lemma 4.1 Let Σ be a Petri net and M_1 and M_2 markings of Σ. Then, the following hold:

(1) for every transition sequence $u \in T^*$, $M_1[u\rangle_\Sigma M_2$ iff $M_2[\overline{u}\rangle_{\overline{\Sigma}} M_1$;

(2) M_2 is reachable from M_1 in Σ iff M_1 is reachable from M_2 in $\overline{\Sigma}$.

Proof (1) can be obtained by induction on the length of u using the fact that $\bar{t}$ undos the effect of t, and (2) follows from (1). □

Now, we can prove the following simple but important result.

Proposition 4.1 Let $\gamma = (\Sigma, M_0)$ be a Petri net and $\overline{M}$ a marking of γ. Then, $\overline{M}$ is a home marking of γ iff $[M_0\rangle_\gamma \subseteq [\overline{M}\rangle_{\overline{\gamma}}$.

Proof Let us suppose first that $\overline{M}$ is a home marking of γ. Then, for every marking $M \in [M_0\rangle_\gamma$ there is a sequence of transitions $v \in T^*$ such that $M[v\rangle_\gamma \overline{M}$. From Lemma 4.1 it follows that $\overline{M}[\overline{v}\rangle_{\overline{\gamma}} M$, which shows that M is reachable from $\overline{M}$ in $\overline{\gamma}$. Therefore, $[M_0\rangle_\gamma \subseteq [\overline{M}\rangle_{\overline{\gamma}}$.

Conversely, let M be a reachable marking in γ. The proposition's hypothesis lead to the fact that M is reachable in $\overline{\gamma}$. Then, from Lemma 4.1 it follows that $\overline{M}$ is reachable from M in γ. Therefore, $\overline{M}$ is a home marking of γ. □

Recall now the space and sub-space inclusion problems as defined in [7] (in what follows, the components of the Petri net Σ_i will be denoted by S_i, T_i, F_1, and W_i, respectively).

The Space Inclusion Problem (SIP)
Instance: $\gamma_1 = (\Sigma_1, M_0^1)$ and $\gamma_2 = (\Sigma_2, M_0^2)$ such that $S_1 = S_2$;
Question: does $[M_0^1\rangle_{\gamma_1} \subseteq [M_0^2\rangle_{\gamma_2}$ hold ?

The Sub-space Inclusion Problem (SSIP)
Instance: $\gamma_1 = (\Sigma_1, M_0^1)$, $\gamma_2 = (\Sigma_2, M_0^2)$, and $S \subseteq S_1 \cap S_2$;
Question: does $[M_0^1\rangle_{\gamma_1}|_S \subseteq [M_0^2\rangle_{\gamma_2}|_S$ hold ?

It is known that both SIP and SSIP are undecidable [7]. Proposition 4.1 leads us to considering the following versions of SIP and SSIP (in what follows $\overline{\gamma}$ is the dual of γ w.r.t. a marking $\overline{M}$ of γ).

The Dual Space Inclusion Problem (DSIP)
Instance: $\gamma = (\Sigma, M_0)$ and a marking $\overline{M}$ of γ;
Question: does $[M_0\rangle_\gamma \subseteq [\overline{M}\rangle_{\overline{\gamma}}$ hold ?,

The Dual Sub-space Inclusion Problem (DSSIP)
Instance: $\gamma = (\Sigma, M_0)$, a marking $\overline{M}$ of γ, and $S' \subseteq S$;
Question: does $[M_0\rangle_\gamma|_{S'} \subseteq [\overline{M}\rangle_{\overline{\gamma}}|_{S'}$ hold ?

From Proposition 4.1 it follows that HMP and DSIP are recursively equivalent and, therefore, DSIP is decidable because HMP is decidable [5].

Definition 4.2 A marking $\overline{M}$ of a Petri net $\gamma = (\Sigma, M_0)$ is called a *home sub-marking of* γ *w.r.t.* $S' \subseteq S$ if for any marking $M \in [M_0\rangle$ there is a marking $M' \in [M\rangle$ such that $M'|_{S'} = \overline{M}|_{S'}$.

The Home Sub-marking Problem (HSMP)
Instance: $\gamma = (\Sigma, M_0)$, a marking $\overline{M}$ of γ, and $S' \subseteq S$;
Question: is $\overline{M}$ a home sub-marking of γ w.r.t. S' ?

Our concept of a home sub-marking is, in fact, the same as that in [5] where it has been proven that the HSMP is decidable. HSMP and DSSIP are not recursively equivalent as HMP and DSIP are. In fact, we shall prove that DSSIP is undecidable for a proper sub-class of Petri nets and, therefore, undecidable for the whole class of Petri nets.

Definition 4.3 A 3-tuple (Σ, s_1, s_2) is called a *two-way Petri net* ($2wPN$, for short) if Σ is a Petri net, s_1 and s_2 are places of Σ, and there is a partition of T, $T = T' \cup T''$, such that ${}^\bullet s_1 = T' = s_1^\bullet$, ${}^\bullet s_2 = T'' = s_2^\bullet$, and $W(s_1, t') = W(t', s_1) = W(s_2, t'') = W(t'', s_2) = 1$ for all $t' \in T'$ and $t'' \in T''$.

Pictuarially, a $2wPN$ is like in Figure 6 (its set of places is $S \cup \{s_1, s_2\}$, where $s_1 \neq s_2$ and $s_1, s_2 \notin S$).

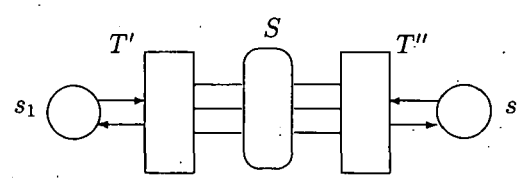


Figure 6: A pictorial view of a two-way Petri net

Theorem 4.1 The dual sub-space inclusion problem for $2wPN$ is undecidable.

Proof We prove the undecidability of DSSIP by reducing SIP to it.

Let γ_1 and γ_2 be an instance of SIP. We consider the $2wPN$ Σ as given in Figure 6, but with the following differences:

- $S = S_1 = S_2$;
- $T' = T_1$ and $T'' = \overline{T}_2$;
- the arcs and their weights between T_1 and S are given by F_1 and W_1, respectively;
- the arcs and their weights between $\overline{T}_2$ and S are given by $\overline{F}_2$ and $\overline{W}_2$, respectively.

Consider then the markings $M_0 = (M_0^1, 1, 0)$ and $\overline{M} = (M_0^2, 0, 1)$, and the marked Petri nets $\gamma = (\Sigma, M_0)$ and $\overline{\gamma} = (\overline{\Sigma}, \overline{M})$.

Thus, we have obtained an instance of DSSIP for $2wPN$ satisfying:

$$[M_0^1\rangle_{\gamma_1} \subseteq [M_0^2\rangle_{\gamma_2} \quad \Leftrightarrow \quad [M_0\rangle_\gamma|_S \subseteq [\overline{M}\rangle_{\overline{\gamma}}|_S.$$

Therefore, SIP is reducible to DSSIP for $2wPN$; the theorem follows then from the undecidability of SIP [7]. □

Clearly, DSSIP for the whole class of Petri nets is undecidable, being undecidable for a sub-class of them.

Conclusions

The existence of home markings is a widely studied subject in the theory of Petri nets [6, 1, 15, 2, 14, 4, 13], but only for very particular classes of them. Thus, in [1] it has been proven that live and 1-safe free-choice Petri nets have home markings. The result has successively been extended to live and safe free-choice Petri nets [15], live and safe equal-conflict Petri nets [14], and deterministically synchronized sequential process systems [11]. All these results make use, more or less directly, of a confluence property which is induced by liveness and safety.

In this paper we have studied the home marking problem for Petri nets. We have proven several results that can be summarized as follows:

- the home marking problem for inhibitor Petri nets is undecidable;
- confluent and notherian Petri nets have an unique home marking;
- the dual sub-space inclusion problem for Petri nets is undecidable.

All these results have been obtained by relating the concept of a home marking to some important concepts in Petri net theory, like confluence, noetherianity, and state space inclusion. Further study of these concepts is, in our opinion, an important subject of research.

Acknowledgement The authors like to thank the referees for their helpful remarks.

References

[1] E. Best, K. Voss. *Free Choice Systems Have Home States*, Acta Informatica 21, 89–100, 1984.

[2] E. Best, L. Cherkasova, J. Desel, J. Esparza. *Characterisation of Home States in Free Choice Systems*, Technical Report 9, Institut für Informatik, Universität Hildesheim, 1990.

[3] R.V. Book, F. Otto. *String Rewriting Systems*, Springer-Verlag, 1993.

[4] J. Desel, J. Esparza. *Free Choice Petri Nets*, Cambridge University Press, 1995.

[5] D. Frutos-Escrig, C. Johnen. *Decidability of Home Space Property*, Tehnical Report LRI 503, 1989.

[6] M. Hack. *Analysis of Production Schemata by Petri Nets*, M.S. Thesis, Project MAC TR-94, Massachusetts Institute of Technology, 1972 (Corrections in Computation Structures Note 17, 1974).

[7] M. Hack. *Decidability Question for Petri Nets*, Technical Report 161, Laboratory for Computer Science, Massachusetts Institute of Technology, 1976.

[8] M. Jantzen. *Conflunet String Rewriting*, Springer-Verlag, 1988.

[9] T. Jucan, F.L. Ţiplea. *Petri Nets. Theory and Application*, Romanian Academy Publishing House, 1999.

[10] M. Minsky. *Recursive Unsolvability of Post's Problem of TAG and other Topics in Theory of Turing Machines*, Annals of Mathematics, vol. 74, no. 3, 1961.

[11] L. Recalde, E. Teruel, M. Silva. *Modeling and Analysis of Sequential Processes that Cooperate through Buffers*, IEEE Trans. Robotics Automat. 14(2). 1998, 267-277.

[12] W. Reisig. *Petri Nets. An Introduction*, Springer-Verlag, 1985.

[13] M. Silva, E. Teruel, J.M. Colom. *Linear Algebraic and Linear Programming Techniques for the Analysis of P/T Net Systems*, Lecture on Petri Nets I: Basic Models, Lecture Notes in Computer Science 1941, 1998, 309–373.

[14] E. Teruel, M. Silva. *Liveness and Home States in Equal Conflict Systems*, Proc. of the 14th International Conference "Application and Theory of Petri Nets", Lecture Notes in Computer Science 691, 1993, 415–432.

[15] W. Vogler. *Live and Bounded Free Choice Nets Have Home States*, Petri Net Newsletter 32, 1989, 18–21.

Received February, 2002

CONTENTS

ISSN 0324—721 X

Felelős szerkesztő és kiadó: Csirik János